DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

CA268 Computer Programming 3 (Data Structure & Algorithms)

Lecture #9 - Sorting and Selection

Dr. Hossein Javidnia

School of Computing
Dublin City University

Email: hossein.javidnia@dcu.ie

2023/2024

In-Place and Stable Sorting Algorithms

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

- A sorting algorithm is **in-place** if the algorithm does not use extra space for manipulating the input but may require a small though nonconstant extra space for its operation. Or we can say, a sorting algorithm sorts in-place if only a constant number of elements of the input array are ever stored outside the array.

- A sorting algorithm is **stable** if it does not change the order of elements with the same value.

Merge-Sort: Divide-and-Conquer

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

The divide-and-conquer pattern consists of the following three steps:

1. **Divide:** If the input size is smaller than a certain threshold (say, one or two elements), solve the problem directly using a straightforward method and return the solution so obtained. Otherwise, divide the input data into two or more disjoint subsets.
2. **Conquer:** Recursively solve the subproblems associated with the subsets.
3. **Combine:** Take the solutions to the subproblems and merge them into a solution to the original problem.

Using Divide-and-Conquer for Sorting

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

We will first describe the merge-sort algorithm at a high level, without focusing on whether the data is an array-based (Python) list or a linked list; we will soon give concrete implementations for each. To sort a sequence S with n elements using the three divide-and-conquer steps, the merge-sort algorithm proceeds as follows:

Using Divide-and-Conquer for Sorting

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

1. **Divide:** If S has zero or one element, return S immediately; it is already sorted. Otherwise (S has at least two elements), remove all the elements from S and put them into two sequences, S_1 and S_2, each containing about half of the elements of S; that is, S_1 contains the first $\lfloor n/2 \rfloor$ elements of S, and S_2 contains the remaining $\lceil n/2 \rceil$ elements.
2. **Conquer:** Recursively sort sequences S_1 and S_2.
3. **Combine:** Put back the elements into S by merging the sorted sequences S_1 and S_2 into a sorted sequence.

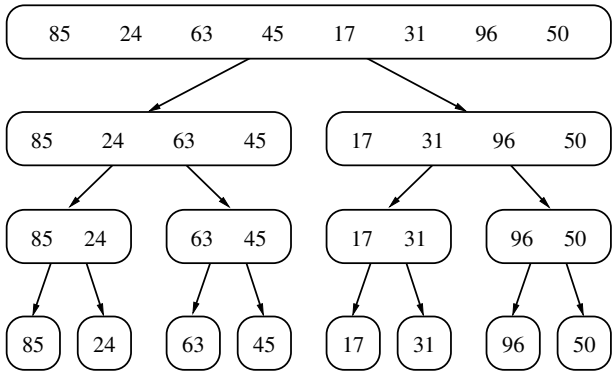
Using Divide-and-Conquer for Sorting

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

In reference to the divide step, we recall that the notation $\lfloor x \rfloor$ indicates the floor of x, that is, the largest integer k, such that $k \leq x$. Similarly, the notation $\lceil x \rceil$ indicates the ceiling of x, that is, the smallest integer m, such that $x \leq m$.

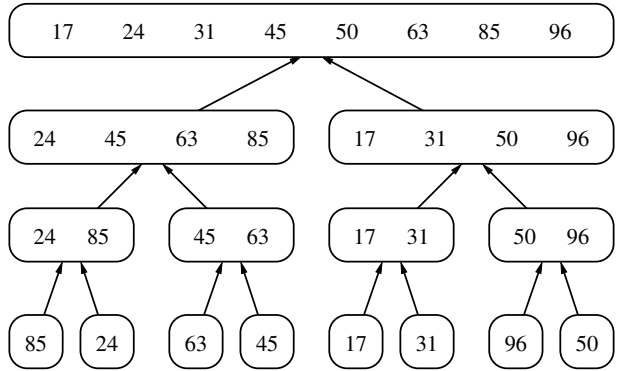
We can visualize an execution of the merge-sort algorithm by means of a binary tree T, called the **merge-sort tree**. Each node of T represents a recursive invocation (or call) of the merge-sort algorithm. We associate with each node v of T the sequence S that is processed by the invocation associated with v. The children of node v are associated with the recursive calls that process the subsequences S_1 and S_2 of S. The external nodes of T are associated with individual elements of S, corresponding to instances of the algorithm that make no recursive calls.

Using Divide-and-Conquer for Sorting



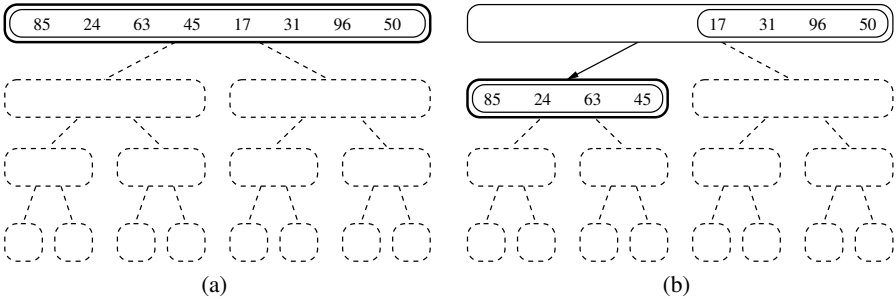
(a)

Using Divide-and-Conquer for Sorting



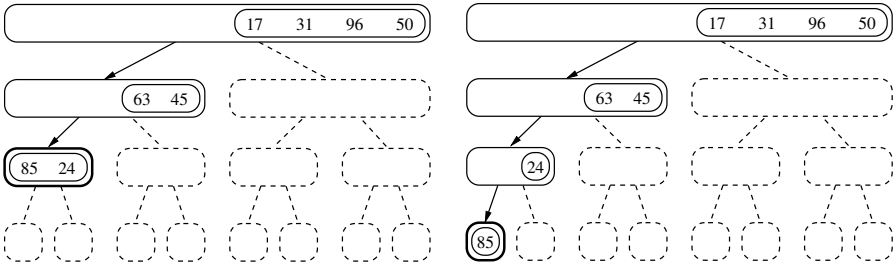
(b)

Using Divide-and-Conquer for Sorting



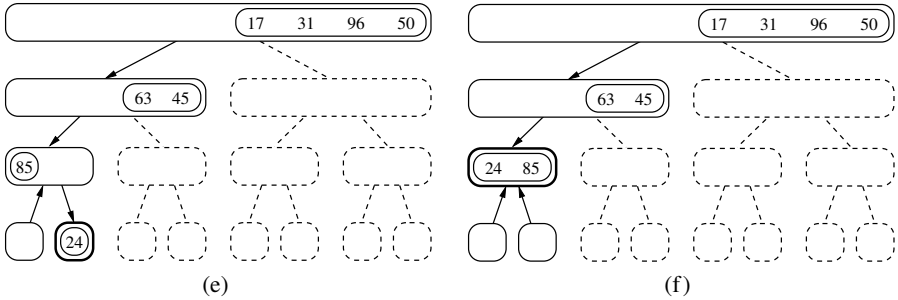
Using Divide-and-Conquer for Sorting

Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University



Using Divide-and-Conquer for Sorting

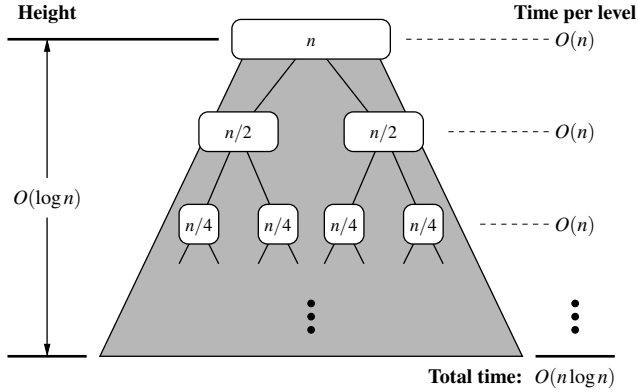
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University



✅ The merge-sort tree associated with an execution of merge-sort on a sequence of size n has height of $\lceil \log n \rceil$.

The Running Time of Merge-Sort

Algorithm merge-sort sorts a sequence S of size n in $O(n \log n)$ time, assuming two elements of S can be compared in $O(1)$ time.



Quick-Sort

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

Like merge-sort, this algorithm is also based on the divide-and-conquer paradigm, but it uses a technique in a somewhat opposite manner, as all the hard work is done before the recursive calls.

The quick-sort algorithm sorts a sequence S using a simple recursive approach. The main idea is to apply the divide-and-conquer technique, whereby we divide S into subsequences, recur to sort each subsequence, and then combine the sorted subsequences by a simple concatenation.

Quick-Sort

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

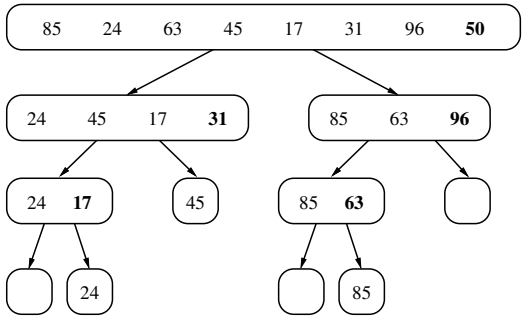
In particular, the quick-sort algorithm consists of the following three steps:

1. **Divide:** If S has at least two elements (nothing needs to be done if S has zero or one element), select a specific element x from S, which is called the pivot. As is common practice, choose the pivot x to be the last element in S. Remove all the elements from S and put them into three sequences:
   - L, storing the elements in S less than x.
   - E, storing the elements in S equal to x.
   - G, storing the elements in S greater than x.

   Of course, if the elements of S are distinct, then E holds just one element the pivot itself.

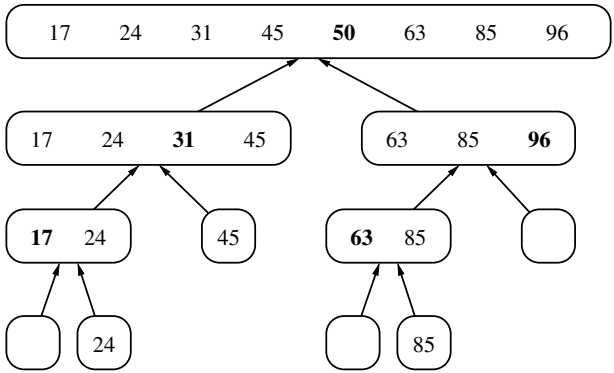
Quick-Sort

2. **Conquer:** Recursively sort sequences L and G.
3. **Combine:** Put back the elements into S in order by first inserting the elements of L, then those of E, and finally those of G.



(a)

Quick-Sort



(b)

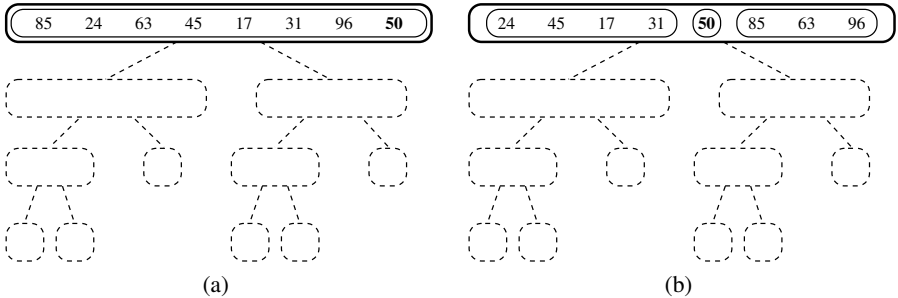
Quick-Sort

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

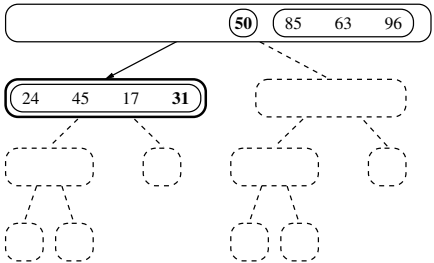
Like merge-sort, the execution of quick-sort can be visualized by means of a binary recursion tree, called the **quick-sort tree**.

Unlike merge-sort, however, the height of the quick-sort tree associated with an execution of quick-sort is linear in the worst case. This happens, for example, if the sequence consists of n distinct elements and is already sorted. Indeed, in this case, the standard choice of the last element as pivot yields a subsequence L of size $n-1$, while subsequence E has size 1 and subsequence G has size 0. At each invocation of quick-sort on subsequence L, the size decreases by 1. Hence, the height of the quick-sort tree is $n-1$.

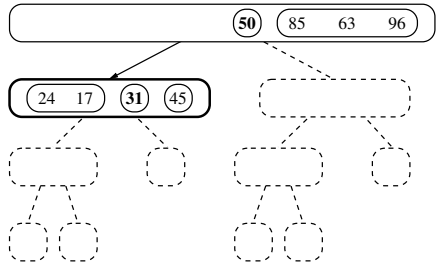
Quick-Sort



Quick-Sort

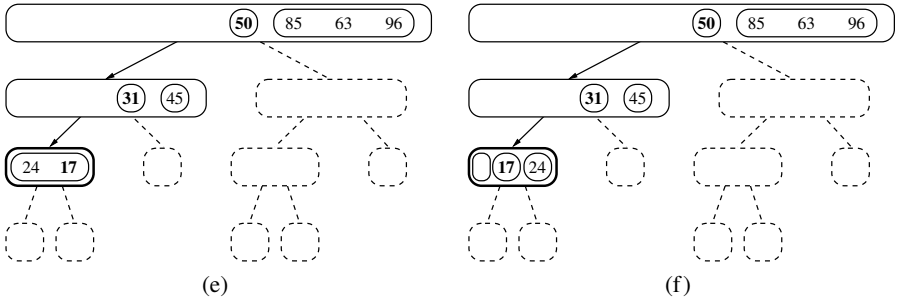


(c)



(d)

Quick-Sort



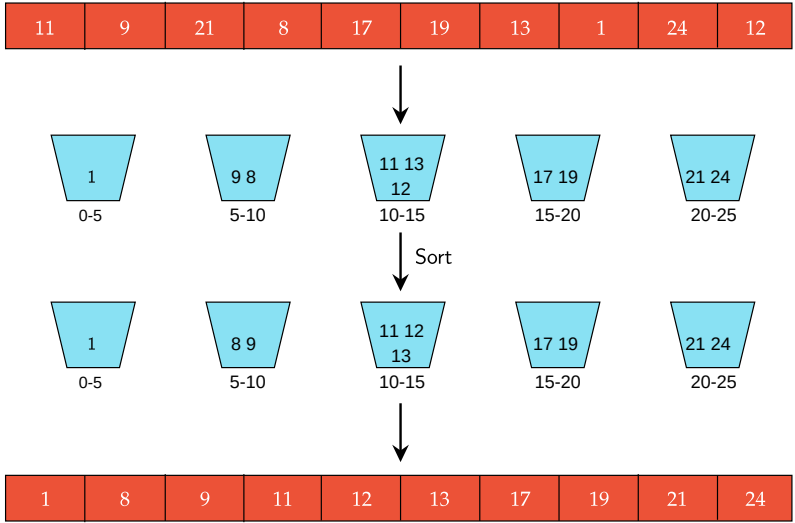
✅ Although its $O(n^2)$-time worst-case performance makes quick-sort susceptible in real-time applications where we must make guarantees on the time needed to complete a sorting operation, we expect its performance to be $O(n \log n)$-time

Bucket Sort

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

Bucket Sort is a sorting algorithm that divides the unsorted array elements into several groups called **buckets**. Each bucket is then sorted by using any of the suitable sorting algorithms or recursively applying the same bucket algorithm. Finally, the sorted buckets are combined to form a final sorted array.

Bucket Sort


11 9 21 8 17 19 13 1 24 12
1
9 8
11 13 12
17 19
21 24
0-5
5-10
10-15
15-20
20-25
Sort
1
8 9
11 12 13
17 19
21 24
0-5
5-10
10-15
15-20
20-25
1 8 9 11 12 13 17 19 21 24

Bucket Sort

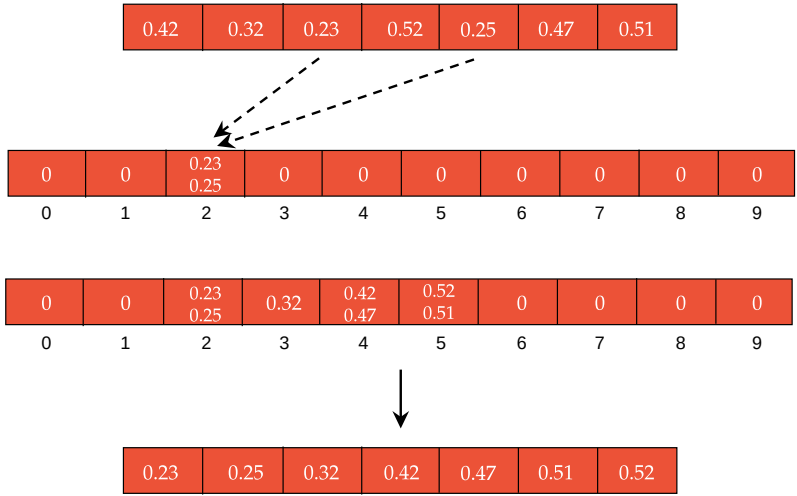
DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

- If the elements of the array are floats ranging between 0 and 1, we primarily make 10 buckets, numbered from 0 to 9, and then insert elements into these buckets depending upon their most significant number. A bucket value is calculated as: $int(elementValue * 10)$.
- If the elements of the array are integers, we simply calculate the range:

$$range = (maxValue - minValue)/noBuckets$$

and divide the whole range into buckets and then perform bucket sorting.

Bucket Sort


0.42
0.32
0.23
0.52
0.25
0.47
0.51
0
0
0.23
0.25
0
0
0
0
0
0
0 1 2 3 4 5 6 7 8 9
0
0
0.23
0.25
0.32
0.42
0.47
0.52
0.51
0
0
0
0
0 1 2 3 4 5 6 7 8 9
0.23
0.25
0.32
0.42
0.47
0.51
0.52

Bucket Sort

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

When there are elements of close range in the array, they are likely to be placed in the same bucket. This may result in some buckets having more number of elements than others.

It makes the complexity depend on the sorting algorithm used to sort the elements of the bucket.

The complexity becomes even worse when the elements are in reverse order. If insertion sort is used to sort elements of the bucket, then the time complexity becomes $O(n^2)$.

Insertion Sort

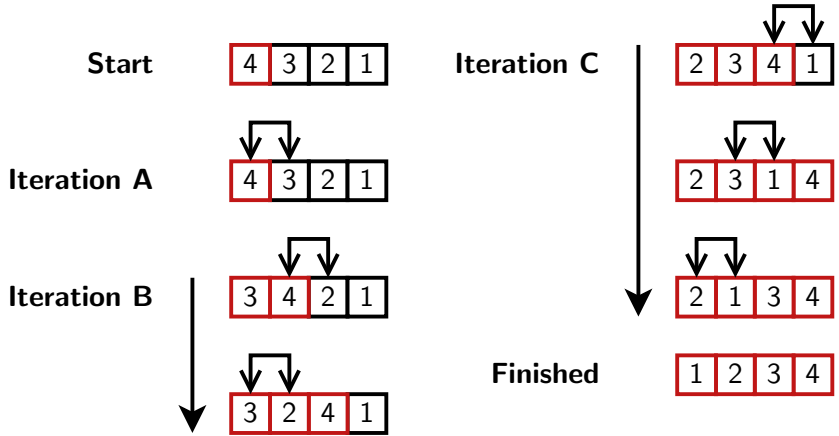
DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

Insertion Sort is one of the simplest sorting techniques which you might have used in your daily lives while arranging a deck of cards.

Assume you have 10 cards, 1 to 10, all shuffled, and you want to sort these cards:

- You would basically pick any random card(e.g. 7), and place it into your left hand, assuming the left hand is meant to carry the sorted cards.
- Then you would pick another random card, say 2, and place 2 in the correct position on your left hand, i.e. before 7.
- Then again if you pick 5, you would place it between 2 and 7 on your left hand, and this way we know we are able to sort our deck of cards. Since we insert one element at a time in its correct position, hence its name "Insertion Sort".

Insertion Sort



Suppose, an array is in descending order. Sorting this array in a ascending order will result in worst case complexity of $O(n^2)$.

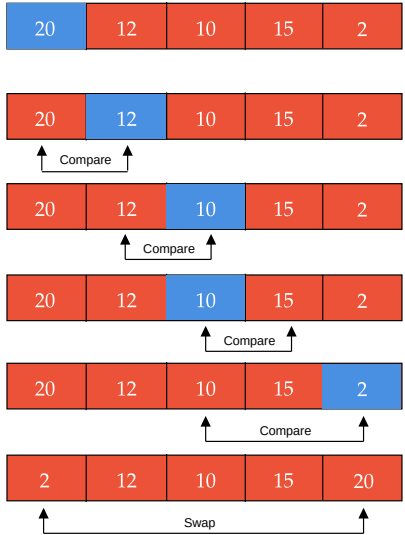
Selection Sort

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

Selection sort is a sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

1. Set the first element as `minimum`.
2. Compare `minimum` with the second element. If the second element is smaller than `minimum`, assign the second element as `minimum`. Compare `minimum` with the third element. Again, if the third element is smaller, then assign `minimum` to the third element otherwise do nothing. The process goes on until the last element.
3. After each iteration, `minimum` is placed in the front of the unsorted list.
4. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.

Selection Sort



The worst case scenario occurs when we want to sort a list in ascending order, but it is arranged in descending order. Time complexity $= O(n^2)$

Bubble Sort

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

Bubble sort, also known as sinking sort, is the easiest sorting algorithm. It works on the idea of repeatedly comparing the adjacent elements, from left to right, and swapping them if they are out-of-order.

In bubble sort, the repetition continues till we get the sorted list. Bubble sort compares all the elements in a list successively and sorts them based on their values.

Bubble Sort

Consider that we want to sort a list in ascending order, here are the steps that the algorithm would follow:

1. Start with the first element.
2. Compare the current element with the next element.
3. If the current element is greater than the next element, then swap both the elements. If not, move to the next element.
4. Repeat steps 1 – 3 until we get the sorted list.

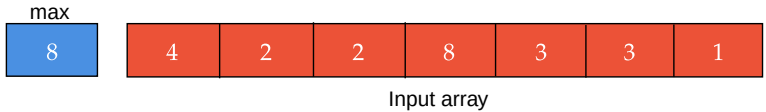
The worst-case occurs when we want to sort a list in ascending order, but it is arranged in descending order. Time complexity $= O(n^2)$

Counting Sort

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

1. Find out the maximum element (let it be `max`) from the given array.



Counting Sort

2. Initialize an array of length `max+1` with all elements 0. This array is used for storing the count of the elements in the array.

Count array

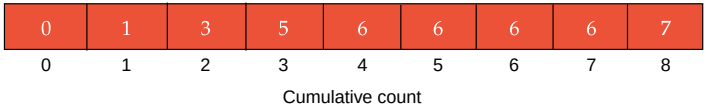
3. Store the count of each element at their respective index in `count` array. For example: if the count of element 3 is 2 then, 2 is stored in the 3rd position of `count` array. If element 5 is not present in the array, then 0 is stored in 5th position.

Count of each element stored

Counting Sort

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

4. Store cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array.

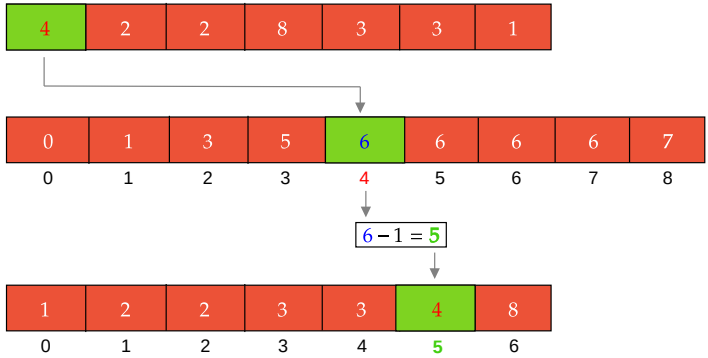


Counting Sort

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

5. Start from the end of the original array. This ensures the stability of the sorting algorithm (meaning that equal elements maintain their relative order).
6. For each element in the original array, look at its value. Use this value as an index in the cumulative sum array.
   - Decrease the value at this index in the cumulative sum array by 1. This tells you the position of this element in the sorted array.
   - Update the cumulative sum array using the decreased value. This adjustment accounts for placing an element of this value and prepares for the next occurrence of the same value.

Counting Sort



Counting Sort

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

The worst case scenario occurs when the array has a very large value for the maximum element of the array. If the size of the array is n and the maximum element of the array is k then, time complexity $= O(n + k)$.

If the maximum value tends to a very large value, the complexity keeps on increasing. Essentially, Counting Sort algorithm will actually not work at all if the value becomes larger than 10^7.

Radix Sort

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

Radix sort is an algorithm that uses counting sort as a subroutine to sort an array of integers/strings in either ascending or descending order. The main idea of radix sort revolves around applying counting sort digit by digit on the given array.

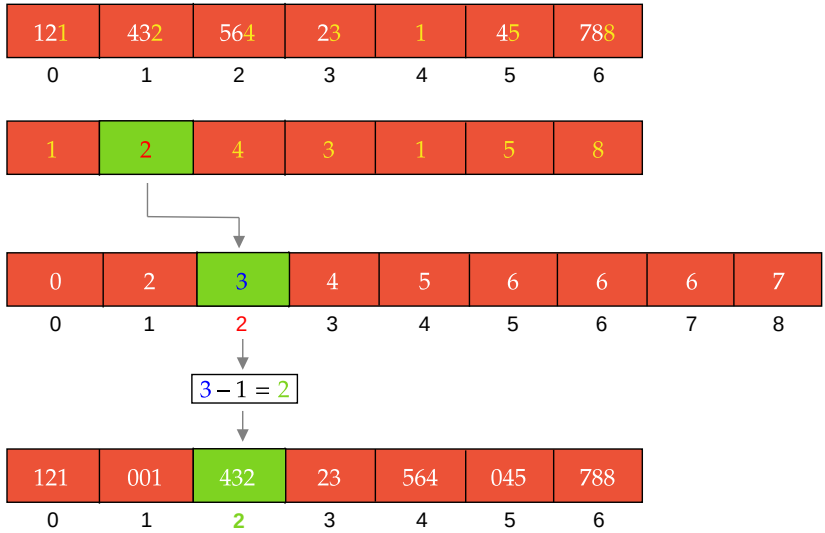
The algorithm sorts the elements by first grouping the individual digits of the same place value. Then, sort the elements according to their increasing/decreasing order.

Radix Sort

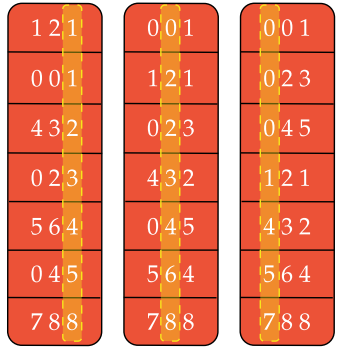
Radix sort steps:

1. Find the largest element in the array, i.e. `max`. Let X be the number of digits in `max`. X is calculated because we have to go through all the significant places of all elements.
2. Go through each significant place one by one. Radix Sort processes digits from the least significant to the most significant. In the next example, we have the largest number 788. It has 3 digits. Therefore, the loop should go up to hundreds place (3 times).
3. Use any stable sorting technique to sort the digits at each significant place. By default counting sort is used for this. Sort the elements based on the unit place digits ($X = 0$)

Radix Sort


121 432 564 23 1 45 788
0 1 2 3 4 5 6
1 2 4 3 1 5 8
0 2 3 4 5 6 6 6 7
0 1 2 3 4 5 6 7 8
3 − 1 = 2
121 001 432 23 564 045 788
0 1 2 3 4 5 6

Radix Sort



In the worst case scenario i.e. array is sorted in reverse order, we need to apply the Counting Sort (an $O(n)$ process) for k times, where k is the number of digits present in the largest element of the array. Therefore, the overall time complexity is $O(n \times k)$.


```
End Lecture #9;
```