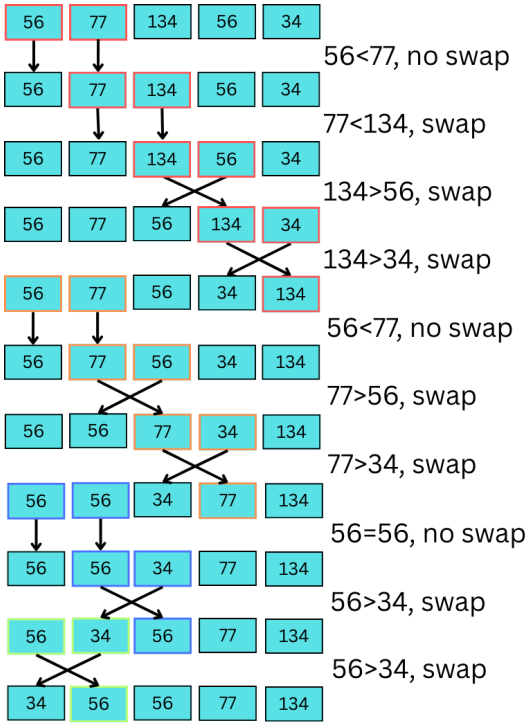
WIA2005 Algorithm Design & Analysis
Tutorial 3

PART 1: Illustration, time complexity and related issues

Illustrate each of the sorting algorithm below. Find the time complexity for each of the algorithm and discuss what is the best application condition of the algorithm. Include example of application for the algorithm.

1. Bubble Sort

A: 56 77 134 56 34



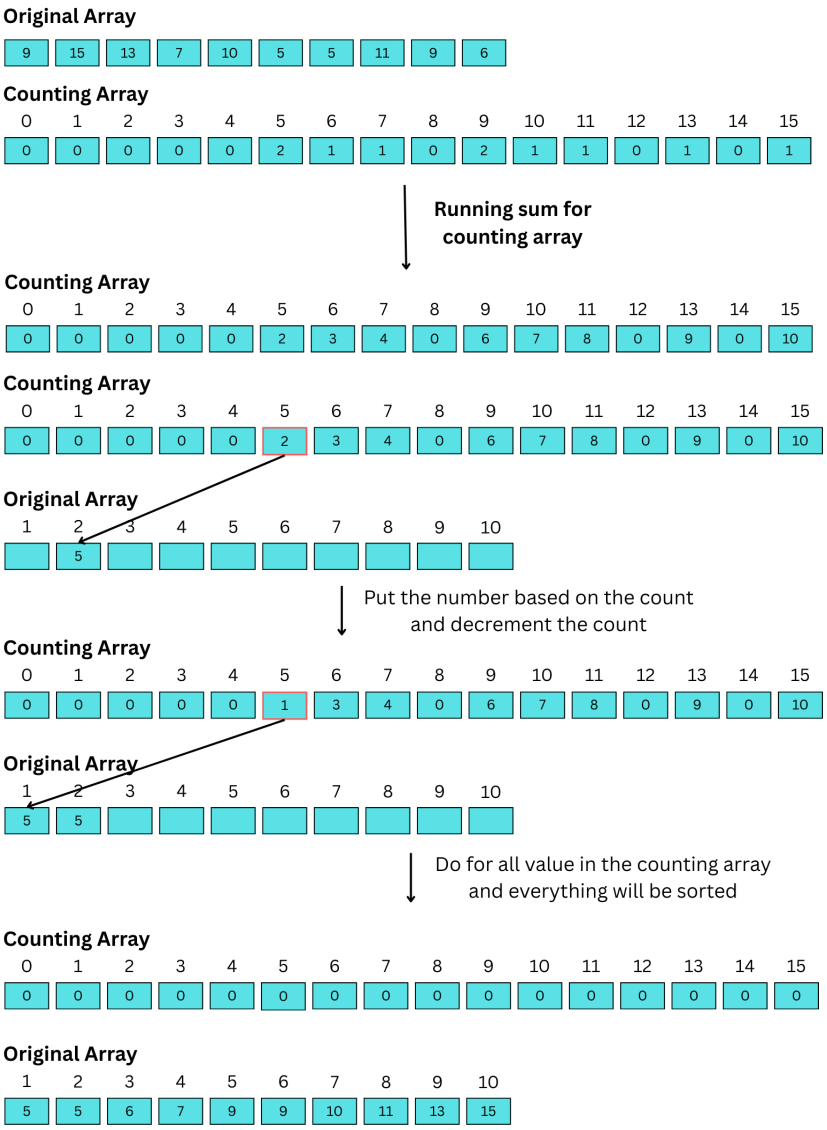
Time Complexity = $O(n^2)$

Best Application Condition:
- Good for a small data set that is almost sorted

Example application:
- Using bubble sort to sort the a list of student names in a class based on their grades

2. Counting Sort

A: 9 15 13 7 10 5 5 11 9 6



Time Complexity : θ(n+k)

n – Size of array

k – Range of numbers in the array

Best Application Condition:

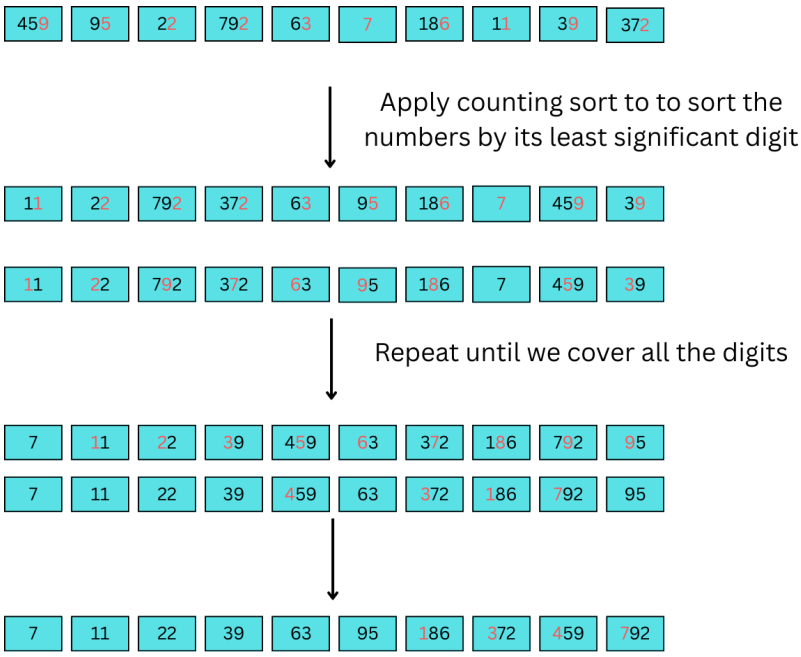
- Best used to sort data in small range

Example application:

- Used to sort data based on digits in a radix sort

3. Radix Sort

A: 459 95 22 792 63 7 186 11 39 372



Time Complexity : θ(nk)

n – Max length of elements

k – Size of array

Best Application Condition:

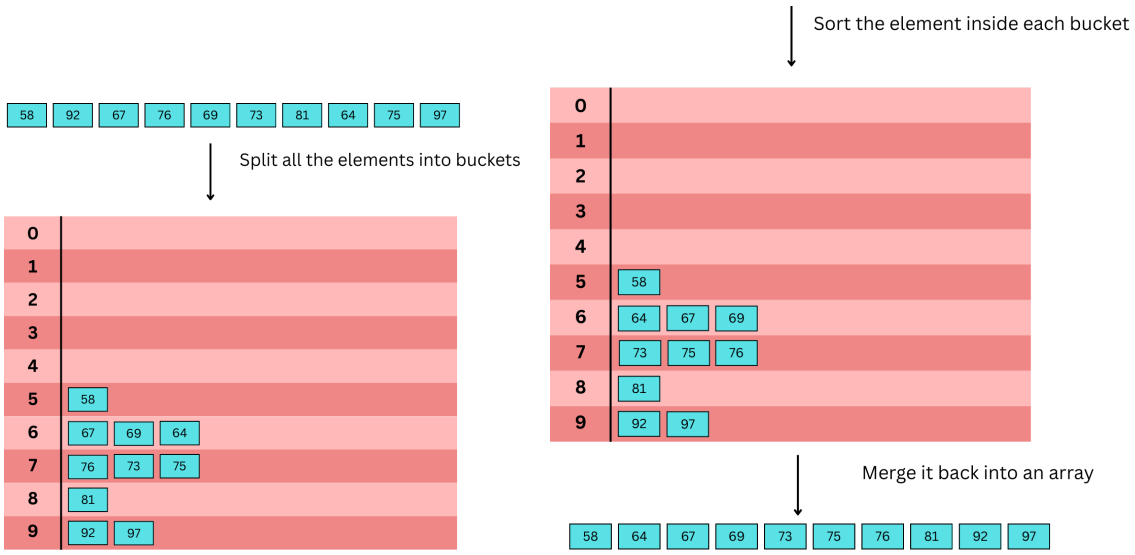
- Best used for a large dataset with a uniform or fixed element length

Example application:

- Used to sort all company staff based on id number

4. Bucket Sort

A: 58 92 67 76 69 73 81 64 75 97



Time Complexity : θ(n)

n – number of elements

(With the assumption that the data is uniformly distributed)

Time Complexity : θ(n+k)

n – number of elements

k – number of buckets

(if number of element and number of bucket is not the same)

Best Application Condition:

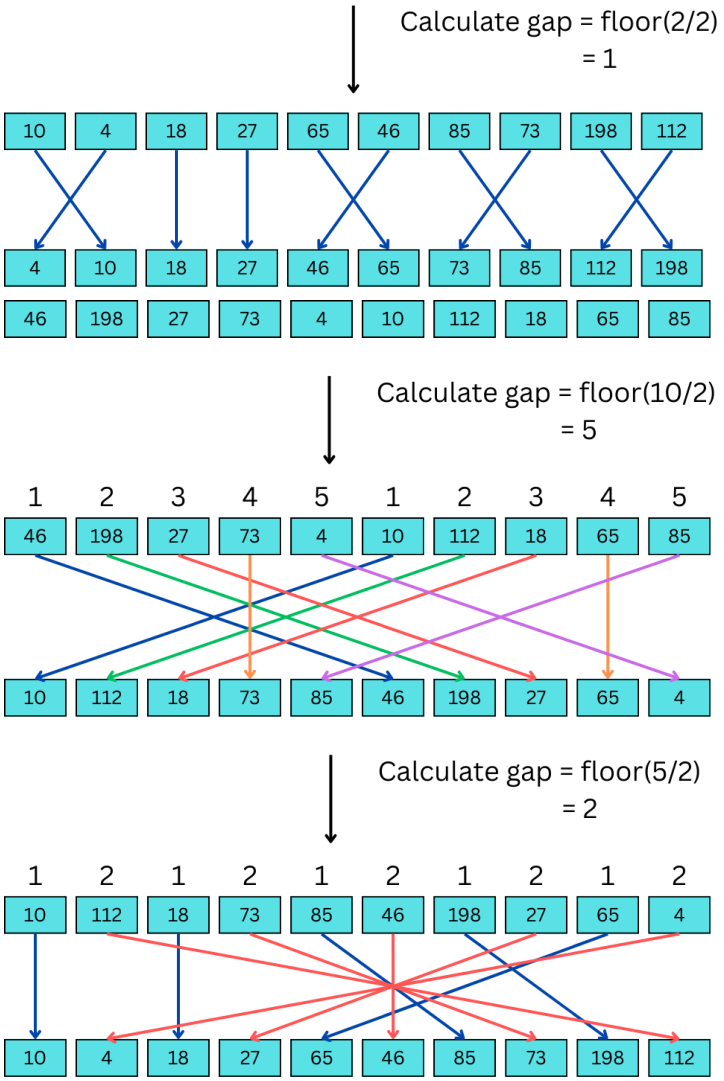
- The data is normally distributed

Example application:

- Used to sort a uniformly distributed floating point numbers

5. Shell Sort

A: 46 198 27 73 4 10 112 18 65 85



Time Complexity : **θ(n log n)**

(Utilize the use of gap to improve from the insertion sort decrease the amount of comparison need)

Best Application Condition:

- Best midsized partially sorted data

Example application:

- Sorting data on a microcontroller where memory is limited

PART 2: Applying the sorting algorithm

Based on the data below, suggest the best algorithm to sort the data. The algorithm suggestion is not limited to the one discussed in the lecture (and please include an additional description).

https://drive.google.com/file/d/1qrqP8g7NTRfb6eZKTDtxi2TT2v7wqced/view?usp=drive_link

1. Data1 = [1000000, 10, 500000, 100, 999999, 1, 750000, 200, 300000, 20]

   Type: Integers

   Size: Small (10 elements)

   Range: Very large (1 to 1,000,000)

   Distribution: Unsorted and sparse over a wide range

   Recommended: Insertion Sort

   For small datasets, insertion sort performs very efficiently due to low overhead.

   Despite the large range, the actual number of elements is tiny, so a complex algorithm isn't needed.

2. Data2 = [3.14, 2.71, 1.41, 0.577, 4.669, 1.618, 2.718, 3.141, 0.693, 2.302]

   Type: Floating point numbers

   Size: Small (10 elements)

   Range: Small (0.577 to 4.669)

   Distribution: Fairly uniform

   Recommended: Bucket Sort

   Ideal for floats in a small, known range and uniformly distributed.

   Each bucket holds a small subrange and is then sorted (often with insertion sort internally).

   Very fast for this kind of data.

3. Data3 = ["banana", "apple", "grape", "cherry", "date"]

   Type: Strings

Size: Small (5 elements)

Lexicographic ordering

Recommended: Insertion Sort

It's simple, easy to implement, and performs well when the list is nearly sorted or small.

String comparisons work perfectly fine with insertion sort (e.g., "apple" < "banana").

It's a **stable sort**, meaning it preserves the original order of equal elements (though not critical here).

4. Data4 = [567, 816, 826, 565, 167, 175, 571, 809, 561, 175, 553, 809, 173, 814, 167]

   Data type: Non-negative integers

   Size: Moderate (15 elements)

   Range: 100 to 900+ (3-digit numbers)

   Duplicates: Present (e.g., 175, 167, 809)

   Recommended: Radix Sort

   Radix Sort is **highly efficient** for sorting fixed-length integers.

   Time complexity is **O(nk)**, where n is the number of elements and k is the number of digits (here, 3).

   It avoids costly comparisons, instead sorting numbers digit by digit (least significant to most).

   Since all values are **non-negative and within a narrow digit range**, Radix Sort works very well.

   More **space-efficient** than Counting Sort in this case, because Counting Sort would require a count array of size ~800.

PART 3: Brief case study

Case Study 1: Employee Shift Scheduling

Introduction:
The efficient allocation of work shifts to employees is a critical aspect of workforce management for many organisations. In this case study, we explore how sorting algorithms were employed to improve the process of scheduling work shifts at a medium-sized retail store. The goal was to reduce labour costs, enhance employee satisfaction, and increase operational efficiency.

Background:
The retail store XYZ Mart had been using a manual shift scheduling process for years. The old method was time-consuming and prone to human errors, resulting in suboptimal labor allocation and employee dissatisfaction. The store aimed to find a solution to minimise overtime costs, ensure proper coverage during peak hours, and consider employee preferences and availability.

Find out:

1. What type of data will be gathered for this shift scheduling system? Create a sample data of size 10. Discuss the possible nature of the data.

   -employee information, employee availability, employee preferences

emp ID	empName	role	avail	shiftPref	hoursWorked	maxHr	Pay/hr (RM)	peak Avail	shift Sat
E01	Afif	Cashier	Mon–Fri	Afternoon	35	40	11	Yes	6
E02	Hazim	Stocker	Tue–Sat	Evening	28	30	9	No	8
E03	Wei Jie	Stocker	Mon–Wed	Evening	25	35	9	No	9
E04	Sitharten	Floor manager	Wed–Sun	Morning	30	40	15	Yes	10
E05	Zannat	Cashier	Mon-Thu	Afternoon	30	35	11	No	10
E06	Sharveena	Cashier	Tue-Sun	Morning	25	25	11	Yes	9
E07	John	Stocker	Wed-Sat	Morning	25	30	9	Yes	8
E08	Daniel	Floor Manager	Mon-Fri	Morning	28	30	15	Yes	9
E09	Sarah	Stocker	Sat–Sun	Afternoon	30	35	9	No	7
E10	Aisyah	Dept Lead	All week	Evening	30	30	18	Yes	10

peakAvail: Peak hour availability, shiftSat: Employee shift satisfaction,

shiftPref: Shift preferences

Nature of data

- Semi-structured Data - includes time ranges, categorical values, and numerical inputs in a flexible but formatted way.
- Dynamic data - fields like **hoursWorked** and **shiftSat** change frequently
- Multidimensional data - multiple factors (preference, availability, wage) influence shift assignment decisions.

2. What sorting algorithm is suitable for the data identified in (1)? Justify your answer.

   Shell sort is the most suitable for the data above.

   Justification:

   1. Doesn't need uniform distribution or limited range - Unlike bucket or counting sort, which rely on specific value ranges
   2. In-place - Does not require extra memory
   3. Flexible - Can be used to sort by any field
   4. Works with complex records - Handles data that isn't just numbers
   5. Efficient for Medium-sized Datasets - Ideal for 10–100 records, like in this case.

   Bubble Sort: Inefficient for anything other than a very small dataset.

   Counting Sort: Not suitable for multi-field employee records.

   Radix Sort: Not practical for categorical + numerical + range-mixed records.

   Bucket Sort: Less suitable for complex employee data.

Case Study 2: Inventory Management for E-Commerce

Introduction:
In the world of e-commerce, efficient inventory management is critical to the success of a business. Sorting plays a pivotal role in organizing products, streamlining order processing, and ensuring customer satisfaction. This case study delves into the application of sorting algorithms in a hypothetical e-commerce company, "ShopSmart," and explores how the choice of sorting algorithm can significantly impact their operational efficiency.

Background:
ShopSmart is an online retailer offering a wide range of products, from electronics to fashion, and experiences high customer traffic. With a vast inventory, the company needs to update frequently and sort products based on various attributes such as price, popularity, and availability.

ShopSmart is experiencing inefficiencies in its inventory management process. Sorting products for display and order fulfilment takes longer than desired, leading to delayed order processing and occasional customer dissatisfaction. The current sorting method is not scalable, and the company is exploring implementing efficient sorting algorithms to optimise their operations.

1. Question1: What type of data will be gathered for this inventory management system? Create a sample data of size 10. Discuss what is the possible nature of the data.

For an inventory management system like the one ShopSmart needs, the data gathered should include attributes relevant to sorting and managing products efficiently. The case study mentions that ShopSmart sorts products based on attributes like price, popularity, and availability, and they need to update these frequently. Therefore, the data will likely include:

- **Product ID**: A unique identifier for each product.
- **Product Name**: The name of the product.
- **Price**: The cost of the product (in dollars, for simplicity).
- **Popularity**: A metric for how popular the product is (e.g., number of views, sales, or a popularity score).
- **Availability**: The stock level or quantity available (e.g., number of units in stock).

These attributes align with the company's need to sort and update products based on the mentioned criteria.

Now, let's create a sample dataset of size 10:

Product ID	Product Name	Price	Popularity	Availability
P001	Laptop	1200	150	10
P002	Smartphone	800	200	5
P003	Headphones	150	80	20
P004	Tablet	600	120	8
P005	Smartwatch	300	90	15
P006	Camera	1000	60	3
P007	Speaker	200	110	12
P008	Monitor	400	70	7
P009	Keyboard	100	50	25
P010	Mouse	50	40	30

This dataset includes 10 products with the relevant attributes that ShopSmart can use for sorting and inventory management.

2. Question2: What sorting algorithm is suitable for the data identified in (1)? Justify your answer.

To determine a suitable sorting algorithm for the dataset, let's analyze the requirements and characteristics of the data and ShopSmart's operations:

- **Data Characteristics**: The dataset is relatively small (sample size of 10, but let's assume in a real-world scenario it could grow to a few thousand products). The attributes (price, popularity, availability) are numerical, and sorting might be needed on different attributes at different times.

- **Operational Needs**: ShopSmart updates products frequently and sorts them based on various attributes. This suggests that sorting might happen multiple times (e.g., sort by price for a sale, then by popularity for a trending section). The current method is not scalable, and they are exploring efficient sorting algorithms.
- **Performance Requirements**: Since this is an e-commerce platform, sorting needs to be fast to ensure a good user experience. Frequent updates also mean the data might not always be fully sorted, so the algorithm should handle partially sorted or unsorted data well.

Given these considerations, let's evaluate a few sorting algorithms:

1. **Bubble Sort**: Simple but inefficient with a time complexity of $O(n^2)$. It's not suitable for ShopSmart because even with a few thousand products, it would be too slow.
2. **Quick Sort**: Has an average time complexity of O(n log n), but its worst-case is $O(n^2)$ if the pivot is poorly chosen. It's in-place and generally fast, but the worst-case scenario might be a concern for frequent sorting.
3. **Merge Sort**: Has a consistent time complexity of O(n log n), but it requires extra space (O(n)) for the merging process. It's stable, which is useful if ShopSmart wants to sort by multiple attributes (e.g., sort by price, then by popularity for equal prices).
4. **Insertion Sort**: Efficient for small datasets ($O(n^2)$ but with a low constant factor) and nearly sorted data (O(n) in the best case). Since ShopSmart updates frequently, the data might often be partially sorted, making this a potential candidate.
5. **Heap Sort**: O(n log n) time complexity and in-place, but not stable. It's less practical for frequent sorting on multiple attributes.

Recommendation: Merge Sort

Justification:

- **Efficiency**: Merge Sort's O(n log n) time complexity ensures good performance even as the dataset grows to a few thousand products. This addresses ShopSmart's scalability concerns.
- **Stability**: Merge Sort is stable, meaning it preserves the relative order of equal elements. This is important for ShopSmart because they might need to sort by multiple attributes (e.g., sort by price, then by popularity for products with the same price). A stable sort ensures the secondary sorting criterion is respected.
- **Frequent Updates**: While Merge Sort requires O(n) extra space, its consistent performance is valuable for frequent sorting operations. Since updates are frequent, the data might not always be sorted, and Merge Sort handles unsorted data well.
- **User Experience**: For an e-commerce platform, fast sorting is critical to avoid delays in displaying products to users. Merge Sort's predictable O(n log n) performance ensures this, even if the dataset grows.

In summary:

While Insertion Sort could be considered for very small or nearly sorted datasets, it doesn't scale well for larger datasets, which ShopSmart might encounter as they grow. Quick Sort, while often faster in practice, has a worst-case scenario that could impact performance, and it's not stable. Merge Sort strikes a balance between efficiency, stability, and reliability, making it a suitable choice for ShopSmart's inventory management system.