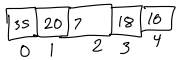
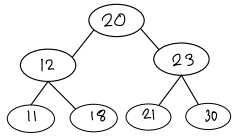
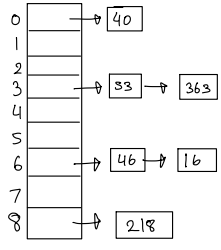
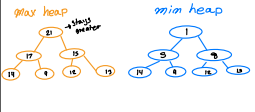
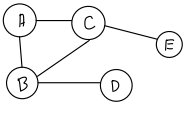
DATA STRUCTURE

11.1 Data Structure

→ Data Structure - way of organizing, storing, and performing operations on data.

Data Structure	Description	Example
Record	- record is a data structure that stores subitems, often called fields, with a name associated with each subitem.	Employee FirstName: Maria LastName: Lu title: Software Developer Salary: 95000
Array	- Array is a data structure stores an ordered collection of elements, where each elements is directly accessible by a position index.	
Best DS for Data insertion → Linked list	- linked list is a DS that stores an ordered list of items in nodes, where each node stores data and has a pointer to next node.	
Binary tree	→ A binary tree is Data Structure in which node stores data and has two children, known as left child and a right child.	
Hash table	→ hash table is a DS that stores unordered items by maping (or hashing) each item to a location in an array.	
Heap	→ Max heap: tree that maintains the simple property that a node's key is greater than or equal to node's children's key. (value of i ≥ value of p) → Min heap: tree that maintains the simple property that a node's key is smaller than or equal to node's children's key. (value of i ≥ value of p)	
Graph	→ graph is DS for representing connections among items, and consists of vertices connected by edges. A vertex represents an item in an graph. An edge represent the a connection between two vertices in a graph.	

1.2 Introduction to Algorithm

→ **Algorithm**: describes a sequence of steps to solve a computational problem or perform a calculation.

→ **A Computational problem**: Specifies an input, a question about the input that can be answered using a computer, and desired output.

Common algorithms

- Longest Common Substring problem: A longest common substring algorithm determines the longest common substring that exists in two input strings.
- Binary Search: The binary search algorithm is an efficient algorithm for searching a sorted array
- Dijkstra's Shortest path: Dijkstra's shortest path algorithm determines the shortest path from a start vertex in a graph.

* NP-Complete problems are set of problem for which no known efficient algorithm exists. Such as:
  - → No efficient algorithm has been found to solve an NP-Complete problem.
  - → No one has proven that an efficient algorithm to solve an NP-complete problem is impossible
  - → if an efficient algorithm exists for one NP-complete problem, then all NP-complete problems can be solved efficiently.

1.3 Relation between DS and Algorithm

* An algorithm is a set of step-by-step instruction to solve a problem, essentially defining the logic for a computation, while a data structure is a specific way of organizing and storing data in a computer's memory, allowing for efficient access and manipulation of that data.

1.4 Recursive definations

Recursive Algorithm → is an algorithm that breaks the problem into smaller subproblems and applies the algorithm itself to solve the smaller subproblems.

↳ a recursive algorithm must have a **base case**: A case where a recursive algorithm completes without applying itself to a smaller subproblem. (base case: end the function).

- Recursive function is a function that call itself. Recursive functions are commonly used to implement recursive algorithms.

Figure 1.41: Sample recursive functions: Factorial(), CumulativeSum(), and ReverseArray()

```
int Factorial(int N) {
  if (N <= 1) {
    return N;
  }
  else {
    return N * Factorial(N - 1);
  }
}
```

```
int CumulativeSum(int N) {
  if (N == 0) {
    return 0;
  }
  else {
    return N + Cumulative(N-1);
  }
}
```

```
Void ReverseArray(int* array, int startIndex,
    int endIndex) {
  if (startIndex >= endIndex) {
    return;
  }
  else {
    std::swap(array[startIndex], array[endIndex]);
    ReverseArray(array, startIndex + 1,
                 endIndex - 1);
  }
}
```

1.5 Algorithm efficiency

→ Algorithm efficiency is typically measured by the algorithm's computational complexity.

→ Computational complexity is the amount of resources used by the algorithm.

Runtime complexity, best case, and worst case

→ An algorithm's runtime complexity is a function, T(N), that represents the number of constant time operations performed by the algorithm on an input of size N.

→ An algorithm's best case is the scenario where the algorithm does the minimum possible number of operations.

→ An algorithm's worst case is the scenario where the algorithm does the maximum of operations.

- Space complexity is a function, S(N) that represents the number of fixed size memory units by the algorithm for an input of size N.

  Ex: The space complexity of an algorithm that duplicates an array of numbers is S(N) = 2N + K, where K is constant representing memory used for things like the loop counter and the two array pointers.

- Auxillary space complexity is the space complexity not including the input data.

  S(N) = N + K

1.6 Constant time operation

→ Constant time operation is an operation that, for a give processor, always operates in the same amount of time, regardless of input values.

→ arithmathic operations - mathematical calculations that combine, transform, or manipulate numbers.

1.7 Growth of functions and Complexity

* N_0 = non negative integer, usually its 1

☆ Upper and Lower bounds

- Lower bound - A function $f(N)$ that is $\leq$ the best case $T(N)$, for all values of $N \geq N_0$
- Upper bound - A function $f(N)$ that is $\geq$ the worst case $T(N)$, for all values of $N \geq N_0$

Growth rates and asymptotic notations

→ Asymptotic notation is the classification of runtime complexity that uses functions that indicate only the growth rate of bounding function.

Such as:

- **O notation (Big O notation)** - provides growth rate for an algorithm's upper bound.
- **Ω notation (Big Omega notation)** - provides a growth rate for an algorithm's lower bound.
- **Θ notation (Big Theta notation)** - provides a growth rate for an algorithm's that is both upper and lower bound.

Table 1.7.1: Notations for algorithm Complexity analysis

Notation	General form	Meaning
O	$T(N)=O(f(N))$	- A positive constant C exists such that, for all $N \geq N_0$, $T(N) \leq C * f(N)$.
Ω	$T(N)=\Omega(f(N))$	- A positive constant C exists such that, for all $N \geq N_0$, $T(N) \geq C * f(N)$.
Θ	$T(N)=\Theta(f(N))$	$T(N)=O(f(N))$ and $T(N)=\Omega(f(N))$

1.8 O notation

→ Big O notation - is a mathematical way of describing how a function (running time of an algorithm) generally behaves in relation to the input size.

Table 1.8.1: Rules for determining Big O notations of Composite functions

$C \cdot O(f(N))$	$O(f(N))$
$C + O(f(N))$	$O(f(N))$
$g(N) \cdot O(f(N))$	$O(g(N) \cdot f(N))$
$g(N) + O(f(N))$	$O(g(N) + f(N))$

Table 1.8.3: Runtime Complexities for various code examples

Notation	Name	Ex. Code

$O(1)$	Constant	`int GetMin(int x, int y){`
		`  if (x < y){`
		`  return x;`
		`  }`
		`  else{`
		`  return y;`
		`  }`
		`}`

continue

O(logN)	Logarithm	int BinarySearch(const int* numbers, int numbersSize, int key) {
		int low = 0;
		int high = numbersSize - 1;
		while (high >= low) {
		int mid = (high + low)/2;
		if (numbers[mid] < key) {
		low = mid + 1;
		}
		else if (numbers[mid] > key) {
		high = mid - 1;
		}
		else return mid;
		}
		return -1;
		}

O(N)	Linear	int LinearSearch (const int* numbers, int numbersSize, int key) {
		for (int i = 0; i < numbersSize; i++) {
		if (numbers[i] == key) {
		return i;
		}
		}
		return -1;
		}

Continue

$O(N \log N)$	Linearithmic	Void MergeSort(int* numbers, int startIndex, int endIndex) {
		if (startIndex < endIndex) {
		int mid = (startIndex + endIndex);
		MergeSort(numbers, startIndex, mid);
		MergeSort(numbers, mid+1, endIndex);
		Merge(numbers, startIndex, mid, endIndex);
		}
		}

$O(N^2)$	Quadratic	Void SelectionSort(int* number, int numbersSize) {
		for(int i=0; i<numbersSize-1; i++) {
		int indexSmallest = i;
		for(int j=1+1; j<numberSize;j++) {
		if(numbers[j] < numbers[indexSmallest]) {
		indexSmallest = j;
		}
		}
		int temp = numbers[i];
		numbers[i] = numbers[indexSmallest];
		numbers[indexSmallest] = temp;
		}
		}

$O(c^N)$	Exponential	int Fibonacci(int N) {
		if((1==N)||(2==N)) {
		return 1;
		}
		return Fibonacci(N-1) +
		Fibonacci(N-2);
		}

1.9 Algorithm analysis

→ Worst-Case runtime: algorithm runtime Complexity for an input that results in the longest execution.

→ if-else statements execute constant number of operations, which has a Big O complexity of O(1)

1.10 Searching and algorithms

→ Algorithm is a sequence of steps for accomplishing a task.

→ Linear search - search algorithm that starts from begining of an array, and checks each element until the search key is found or the end of the array is reached.

- ✯ Worst case: as list sizes grow O(N)
- ✯ Best case: O(1)

Figure 1.10.1: Linear Search algorithm

```
#include <iostream>
Using namespace std;

int LinearSearch (const int* numbers, int numbersSize, int key){
    for (int i=0; i<numbersSize; i++){
        if (numbers[i] == key){
        return i;
    }
  }
  return -1;
}
```

1.11 Binary Search

* Linear Search require searching through all elements which can lead to long runtimes.

Binary search → a search algorithm that finds the position of target value in a sorted list.
↳ How it works : → starts in the middle
↳ if key > middle then proceeds right
↳ else proceeds left.
repeats

Figure 1.11.1: Binary search algorithm

```
#include <iostream>
Using namespace std;

int BinarySearch(const int* numbers, int numbersSize, int key){
   int low = 0;
   int high = numbersSize - 1;

  while(high >= low){
    int mid = (high + low)/2;
    if(numbers[mid] < key){
       low = mid +1;
     }
     else if (numbers[mid] >key){
       high = mid -1;
      }
      else{
       return mid;
      }
   }
  return -1;
}
```

→ Binary Search efficiency: $\lceil \log_2 N \rceil + 1$ Ex: $\lceil \log_2 32 \rceil + 1 = 6$

1.12 Sorting: Introduction

*Sorting → Process of rearranging collection of elements either into ascending (or descending) order.

1.13 Bubble Sort (watch youtube video) (not also suitable for large dataset)

*Bubble Sort → a basic algorithm that arranges a list of numbers or other elements in order by comparing adjacent elements and swapping them if they are out of order.

Figure 1.13.1: Bubble Sort algorithm

```
Void BubbleSort (int* numbers, int numbersSize) {
    for (int i = 0; i < numbersSize - 1; i++) {
        for (int j = 0; j < numbersSize - i - 1; j++) {
            if (numbers[j] > numbers[j + 1]) {
                int temp = numbers[j];
                numbers[j] = numbers[j + 1];
                numbers[j + 1] = temp;
            }
        }
    }
}
```

Worst case: $O(N^2)$

1.14 Selection Sort (watch youtube video) (inefficient for large dataset)

*Selection Sort - Sorting algorithm that treats the input as two parts, a sorted part and unsorted part, and repeatedly selects the minimum value to move from the unsorted part to the end of the sorted part.

continues →

Code

```
#include <iostream>
#include <string>
using namespace std;

Void SelectionSort (int* numbers, int numbersSize) {
    for (int i=0; i < numbersSize - 1; i++) {
        // Find index of smallest remaining element
        int indexSmallest = i;
        for (int j=i+1; j < numbersSize; j++) {        // compares every thing
            if (numbers[j] < numbers[indexSmallest]) {
                indexSmallest = j;
            }
        }
        int temp = numbers[i]; // Swap numbers[i] and numbers[indexSmallest]
        numbers[i] = numbers[indexSmallest];
        numbers[indexSmallest] = temp;
    }
}
```

2 < 4
i=2

```
String ArrayToString (int* array, int arraySize) {
    // Special case for empty string
    if (0 == arraySize) {
        return String ("[]");
    }
    String result = "[" + to_String (array [0]); // Start the String with opening '[' and element 0
    for (int i=1; i < arraySize; i++) {
        result += ", "; // For each remaining element, append comma, space, and element
        result += to_String (array[i]);
    }
    result += "]"; // Add closing ']' and return result
    return result;
}
```

```
int main (){
    int numbers[] = { 10, 2, 78, 4, 45, 32, 7, 11 3};
    int numbersSize = sizeof (numbers) / sizeof (numbers[0]);
    cout << "Unsorted: " << ArrayToString (numbers, numbersSize) << endl;
    SelectionSort (numbers, numbersSize);
    Cout << "Sorted: " << ArrayToString (numbers, numberSize ) << endl;
return 0;
}
```

1.15 Insertion Sort

(watch youtube video)

(good for nearly sorted arrays

☆ Insertion Sort - Sorting algorithm that treats the input as two parts, a sorted and unsorted part, and repeatedly inserts the next value from the unsorted part into the correct location in the sorted part

Code

runtime: $O(N^2)$

```
for (int i=1; i< numbersSize; i++) {
    int j=i;
    // Insert numbers[i] into Sorted part
    // stopping once numbers[j] in Correct position
  while (j>0 && numbers[j] < numbers[j-1]){
    // Swap numbers[j] and numbers[j-1]
    int temp = numbers[j];
    numbers[j] = numbers[j-1];
    numbers[j-1] = temp;
    j--;
  }
}
```

→ insertion sort best case: ascending order
→ insertion sort case case: descending order

→ A nearly sorted array only contains element not in Sorted Order ∴ Insertion Sort is good for this sorting algorithm.

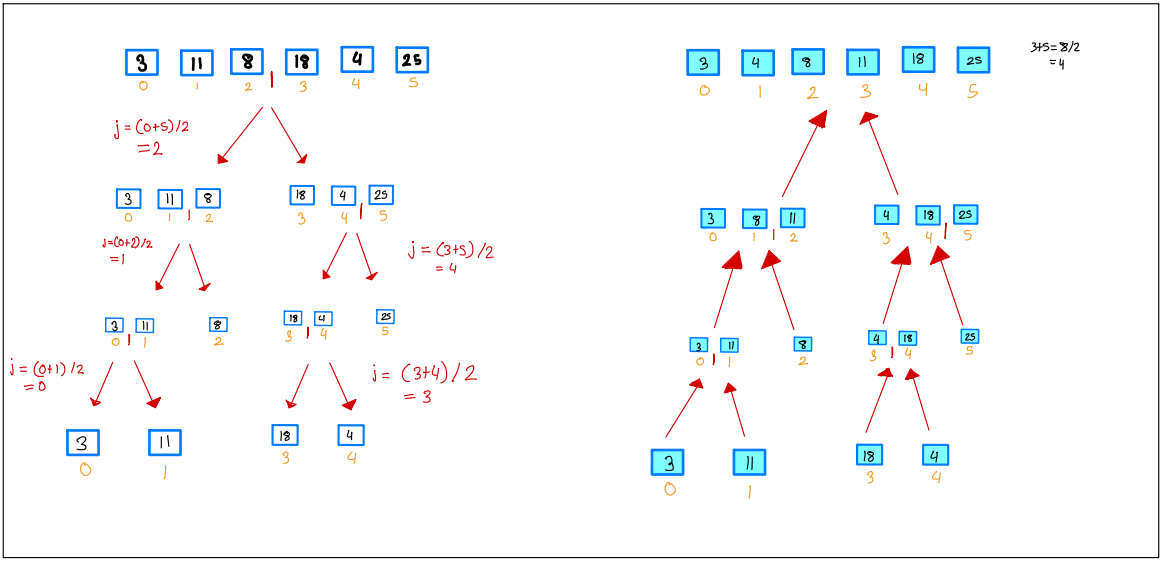
Runtime: $O((N-C)*1+(C*N)) = O(N)$.

1.16 Merge Sort

(Watch Youtube videos) (performs poorly on small arrays)

→ Merge Sort – Sorting algorithm that divides an array into two halves, recursively sorts each half, and then merges the sorted halves to produce a sorted array.

Visual Representation



Code

```
Void Merge (int * numbers, int leftFirst, int leftLast, int rightLast) {
    int mergedSize = rightLast - leftFirst + 1; // mergedSize is 6
    int * mergedNumbers = new int [mergedSize]; // creates another new array that is size of 6
    int mergePos = 0;
    int leftPos = leftFirst;
    int rightPos = leftLast + 1;

    While (leftPos <= leftLast && rightPos <= rightLast) {
        if (numbers [leftPos] <= numbers [rightPos]) {
        mergedNumbers [mergePos] = numbers [leftPos];
        leftPos ++;
        }
        else {
        mergedNumbers [mergePos] = numbers [rightPos];
        rightPos ++;
        }
        mergePos ++;
    }
```

AC continues.

→ Code Continues

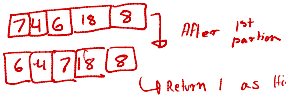
```
While (leftPos <= leftLast) {
    mergedNumbers[mergePos] = number[leftPos];
    leftPos++;
    mergePos++;
}
While (rightPos <= rightLast) {
    mergedNumbers[mergePos] = numbers[rightPos];
    rightPos++;
    mergePos++;
}
for (mergePos = 0; mergePos < mergedSize; mergePos++) {
    number[leftFirst + mergePos] = mergedNumbers[mergePos];   → Copy back to original array
}
delete[] mergedNumbers;                                       → delete the array that was created
}
```

* The runtime of merge sort is O(N log N)
  ↳ Merge sort requires O(N) additional memory elements for temporary array of merged elements.

Q: How many recursive partioning levels required for an array of 8 elements?

8/2 = 4/2 = 2/2 = 1
① ② ③

$\log_2(8) =$



↳ best sorting algo.

1.17 Quick Sort

(Watch Youtube video) (bad when partioning picks smallest or largest element as the pivot element every time)

Runtime O(N log N)

Quick Sort – Sorting algorithm that repeatedly partitions the input low and high parts (each part unsorted), then recursively sorts each part.

↳ To partition, quicksort chooses a pivot to divide the data into low and high parts.

↳ Pivot – Can be any value within the array being sorted and is commonly the middle element's value.

```
int Partition (int* numbers, int lowIndex, int highIndex) {
    int midpoint = lowIndex + (highIndex - lowIndex)/2; // Pick middle element as the pivot
    int pivot = numbers[midpointI];

    bool doe = false;
    while (!done) {
      while (numbers[lowIndex] < pivot) {
        lowIndex++;
      }
      while (pivot < numbers[highIndex]) {
        highIndex--;
      }
      // If lowIndex have met or crossed each
      // other, then partitioning is done
      if (lowIndex >= highIndex) {
        done= true;
      }
      else { // Swap numbers[lowIndex] and numbers[highIndex]
          int temp = numbers[lowIndex];
          numbers[lowIndex] = numbers[highIndex];
          numbers[highIndex] = temp;
          lowIndex++;
          highIndex--;
      }
    }
    return highIndex;
}
```

* Pivot = $\log_2(N)$
* if largest element as pivot N-1

↳ additional function known as partition

QuickSort is Recursive function

```
Void QuickSort(int* numbers, lowIndex, highIndex){
    if (highIndex <= lowIndex) {
      return;
    }
    int lowEndIndex = Partition(numbers, lowIndex, highIndex);
    Quicksort(numbers, lowIndex, lowEndIndex);
    Quicksort(numbers, lowEndIndex+1, highIndex);
}
```

→ How many levels are required for array of 1024 elements? (3-1) 1024-1 = 1023

→ How many total calls to QuickSort() are made to sort an array of 1024 elements? 1+2·(1024-1) = 2047 calls

1.18 Bucket Sorting

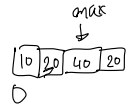
(Watch Youtube) (Worst case: negative #'s and when one bucket gets all the elements)

* Bucket Sorting- is a sorting algorithm that works by distributing the elements of an array into a number of buckets.

Figure 1.18.1: Bucket sort algorithm

Code

```
Void BucketSort(vector<int>& numbers, int bucketCount) {
    if (numbers.size() < 1) {
        return;
    }
    // Create buckets
    vector<vector<int>> buckets(bucketCount);
    // Find the max value
    int maxValue = numbers[0];
    for (int i = 1; i < numbers.size(); i++) {
        if (numbers[i] > maxValue) {
            maxValue = numbers[i];
        }
    }
    // Put each number in a bucket
    for (int i = 0; i < numbers.size(); i++) {
        int bucketIndex = numbers[i] * bucketCount / (maxValue + 1);
        buckets[bucketIndex].push_back(numbers[i]);
    }
    // Sort each bucket
    for (int bucketIndex = 0; bucketIndex < bucketCount; bucketIndex++) {
        vector<int>& bucket = buckets[bucketIndex];
        sort(bucket.begin(), bucket.end());
    }
    // Combine all buckets back into numbers vector
    int outputIndex = 0;
    for (int bucketIndex = 0; bucketIndex < bucketCount; bucketIndex++) {
        vector<int>& bucket = buckets[bucketIndex];
        numbers[outputIndex] = bucket[i];
        outputIndex++;
    }
}
```



1.19 Radix Sort

(Watch Youtube videos) (Worst case: When all elements have the same number of digits except one)

* Radix Sort algorithm is designed for integers,
  - a type of bucket sort

- A Bucket is a collection of integer values that all share a particular digit value such as 44, 14, 54 — all have 4 as the 1's digit, and would all be placed into bucket 4 when subdividing by the 1's digit.

* Radix Sort → an algorithm that sorts data by grouping the keys by individual digits that share the same significant position and value (place value).

Code

```
Void RadixSort (int* array, int arraySize) {
    vector<vector<int>> buckets(10);
    int digitIndex;
    int maxDigits = RadixGetMaxLength(array, arraySize);

    // Start with the least significant digit
    int pow10 = 1;
    for (digitIndex = 0; digitIndex < maxDigits; digitIndex++) {
        for (i = 0; i < arraySize; i++) {
            int bucketIndex = abs(array[i]/pow10) % 10;
            buckets[bucketIndex].push_back(array[i]);
        }
        int copyBackIndex = 0;
        for (int i = 0; i < 10; i++) {
            for (j = 0; j < buckets[i].size(); j++) {
                array[copyBackIndex] = buckets[i].at(j);
                copyBackIndex++;
            }
            buckets[i].clear();
        }
        pow10 *= 10;
    }
}
```

* Dont need this

* How they differ? (highlighted)

Figure 1.192 Radix Sort algorithm (for negative and non negatives)

Continues

```
// Return the length, in number of digits,
// of an integer value
int RadixGetLength(int value) {
    if (value == 0) {
        return 1;
    }
    int digits = 0;
    while (value != 0) {
        digits++
        value /= 10;
    }
    return digits;
}

// Returns the max length, in number of digits,
// out of all array elements
int RadixGetMaxLength(const int* numbers, int numbersSize){
    int maxDigits = 0;
    for (int i=0; i < numbersSize; i++) {
        int digitCount = RadixGetLength(numbers[i]);
        if (digitCount > maxDigits) {
            maxDigits = digitCount;
        }
    }
    return maxDigits;
}
```

*Still need this

```
Void RadixSort (int* numbers, int numbersSize) {
    vector<vector<int>> buckets(10);
    int copyBackIndex;
    int maxDigits = RadixGetMaxLength(numbers, numbersSize);
    int pow10 = 1;
    for (int digitIndex = 0; digitIndex < maxDigits; digitIndex++){
        for(int i=0; i < numbersSize; i++) {
            int num = numbers[i];
            int bucketIndex = (abs(num)/pow10)%10;
            buckets[bucketIndex].push_back(num);
        }
        copyBackIndex = 0;
        for (int i = 0; i < 10; i++) {
            vector<int>& bucket = buckets[i];
            for(int j= 0; j < bucket.size(); j++) {
                array[copyBackIndex] = bucket[j];
                copyBackIndex++;
            }
            buckets[i].clear();
        }
        pow10 *= 10;
    }
}
```

Continues

```
vector<int> negatives;
vector<int> nonNegatives;
for (int i=0; i < numbersSize; i++) {
    int num= numbers[i];
    if (num < 0) {
        negatives.push_back(num);
    }
    else {
        nonNegatives.push_back(num);
    }
}
```

Continues

```
// Copy sorted content to array -
// negatives in reverse, then
// non-negatives
copyBackIndex = 0;
for (int i = negatives.size()-1; i >= 0; i--){
    numbers[copyBackIndex] = negatives[i];
    copyBackIndex++;
}
for (int i = 0; i < nonNegatives.size(); i++) {
    numbers[copyBackIndex] = nonNegatives[i];
    copyBackIndex++;
}
```

Runtime = O(n)

1.20 Overview of fast sorting algorithms

* A fast sorting algorithm is a sorting algorithm that has a average runtime complexity of O(N log N), or better

Such as O(log N), O(1)

Sorting Algorithm	Avg case runtime Complexity	Fast?
- Selection Sort	$O(N^2)$	No
- Insertion Sort	$O(N^2)$	No
- Shell Sort	$O(N^{1.5})$	No
- Quick Sort	O(N log N)	Yes
- Merge Sort	O(N log N)	Yes
- Heap Sort	O(N log N)	Yes
- Radix Sort	O(N)	Yes

* An element sorting algorithm that operates on an array of elements that can be compared to each other. ex: (Strings & more)

Sorting Algorithm	Comparison?
Selection Sort	Yes
Insertion Sort	Yes
Shell Sort	Yes
Quick Sort	Yes
Merge Sort	Yes
Heap Sort	Yes
Radix Sort	No

Best and Worst case runtime Complexity

Sorting Algorithm	Best case runtime Complexity	Avg case runtime Complexity	Worst case runtime Complexity
Quicksort	O(N log N)	O(N log N)	$O(N^2)$
Merge Sort	O(N log N)	O(N log N)	O(N log N)
Heap Sort	O(N)	O(N log N)	O(N log N)
Radix Sort	O(N)	O(N)	O(N)