Stack.

```
template <class T>
class Node {
public:
    T data;
    Node<T>* next;

    Node() : next(nullptr) {}
    Node(const T& data) : data(data), next(nullptr) {}
};
```

```
template<class T>
class Stack {
    public:
     Stack();
     ~Stack(); // destructor
     void push(const T& newEntry); // adds an element to top of stack
     void pop(); // removes element from top of stack
     T top() const; // returns a copy of element at top of stack
     bool isEmpty() const; // returns true if no elements on stack false otherwise
     private:
     Node<T>* top_; // Pointer to top of stack
     int item_count; // number of items currently on the stack

     }; //end Stack
```

```
template<class T>
void Stack<T>::push(const T& new_entry) {
    Node<T>* new_node = new Node<T>(new_entry);
    new_node->next = top_;
    top_ = new_node;
    item_count++;
}
```

```
template<class T>
void Stack<T>::pop() {
    if (isEmpty()) {
      throw std::runtime_error("Pop attempted on an empty stack.");
    }
    Node<T>* node_to_delete = top_;
    top_ = top_->next;
    delete node_to_delete;
    item_count--;
}
```

Queues.

```
template<class T>
void Queue<T>::enqueue(const T& new_entry) {
    Node<T>* new_node = new Node<T>(new_entry);
    if (isEmpty()) {
        front_ = newNode;
    } else {
        back_->back = newNode;
    }
    back_ = new_node;
    item_count++;
}
```

```
template<class T>
void Queue<T>::dequeue() {
    if (isEmpty()) {
      throw std::runtime_error("Pop attempted on an empty stack.");
    }
     Node<T>* node_to_delete = front_;
     front = front -> back_;
     delete node_to_delete;
     item_count--;
     if (isEmpty()) {
        back_ = nullptr;
    }

}
```

selection sort.

vector<int> arr = {43,11,5,9,1,2,9,12,34,70};

1,11,5,9,43,2,9,12,34,70
1,2,5,9,43,11,9,12,34,70
1,2,5,9,43,11,9,12,34,70
1,2,5,9,9,11,43,12,34,70
1,2,5,9,9,11,43,12,34,70
1,2,5,9,9,11,12,43,34,70
1,2,5,9,9,11,12,34,43,70
1,2,5,9,9,11,12,34,43,70
1,2,5,9,9,11,12,34,43,70

```
void selectionSort(std::vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; ++i) {
        int minIndex = i;
        for (int j = i + 1; j < n; ++j) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        std::swap(arr[i], arr[minIndex]);
        printArray(arr);
    }
```

Smart pointer ownership = object's destructor
automatically invoked when pointer goes out of scope or set to nullptr
3 types:
- shared_ptr - keeps track of #
of pointers to one object. The last
one must delete object
- unique_ptr - only smart
pointer allowed to point to the object
- weak_ptr - Points but does not own

```
int binarySearch(const std::vector<int>& arr, int
target) {
    int left = 0;
    int right = arr.size() - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid;
        }
        if (target < arr[mid]) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    // If we reach here, the element was not
present
    return -1;
}
```

```
std::shared_ptr<int> ptrA = std::make_shared<int>(10);
std::shared_ptr<int> ptrB = std::make_shared<int>(20);

std::weak_ptr<int> weakPtrA = ptrA;
std::weak_ptr<int> weakPtrB = ptrB;
cout << weakPtrA.lock() << endl; // adress of shared pointer
cout << *weakPtrA.lock() << endl; // the value of shared pointer to which weakPtrA is pointed to

std::weak_ptr<int> weakPtrC = weakPtrB; // weak pointer pointed to another weak pointer
cout << *weakPtrC.lock() << endl; // print 20 the value to which pointerB is pointed

// Creating a unique_ptr
std::unique_ptr<MyClass> uniquePtr = std::make_unique<MyClass>();
// Using the unique_ptr
uniquePtr->DoSomething();
uniquePtr _ptr<int> uniquePtrB = uniquePtr; // wrong

std::unique_ptr<int> ptr1 = std::make_unique<int>(10);
std::unique_ptr<int> ptr2 = std::move(ptr1);  // Transfer ownership to ptr2
```

```
// insertion sort
43,11,5,9,1,2,9,12,34,70
// 11,43,5,9,1,2,9,12,34,70
// 5,11,43,9,1,2,9,12,34,70
// 5,9,11,43,1,2,9,12,34,70
// 1,5,9,11,43,2,9,12,34,70
// 1,2,5,9,11,43,9,12,34,70
// 1,2,5,9,9,11,43,12,34,70
// 1,2,5,9,9,11,12,43,34,70
// 1,2,5,9,9,11,12,34,43,70
// 1,2,5,9,9,11,12,34,43,70
// 1,2,5,9,9,11,12,34,43,70
```

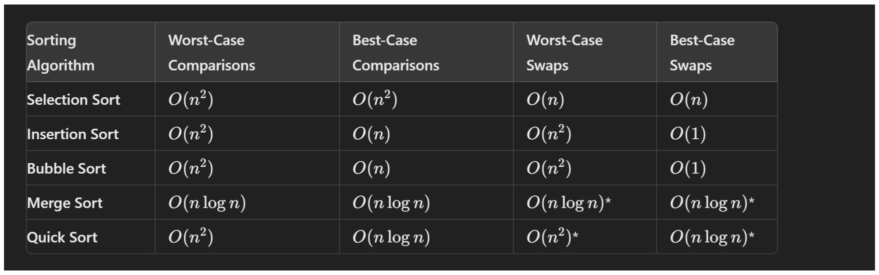
given array - [1,32,42,2,34,11,81,0,2]

Bubble sort .

1) 1,32,2,34,11,42,0,2,81
2) 1,2,32,11,34,0,2,42,81
3) 1,2,11,32,0,2,34,42,81

merge sort

Merge Sort
Initial Array: [1, 32, 42, 2, 34, 11, 81, 0, 2]
[1, 32, 42, 2, 34] and [11, 81, 0, 2]
[1, 32, 42] and [2, 34] and [11, 81] and [0, 2]
[1, 32] and [42] and [2] and [34] and [11] and [81] and [0] and [2]
[1] and [32] and [42] and [2] and [34] and [11] and [81] and [0] and [2]
[1, 32] and [2, 42] and [11, 34] and [0, 2, 81]
[1, 2, 32, 42] and [11, 34, 0, 2, 81]
[1, 2, 11, 32, 34, 42] and [0, 2, 81]
[0, 1, 2, 2, 11, 32, 34, 42, 81]



Sorting Algorithm	Worst-Case Comparisons	Best-Case Comparisons	Worst-Case Swaps	Best-Case Swaps
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n)$
Insertion Sort	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$
Bubble Sort	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$*	$O(n \log n)$*
Quick Sort	$O(n^2)$	$O(n \log n)$	$O(n^2)$*	$O(n \log n)$*

Binary search tree

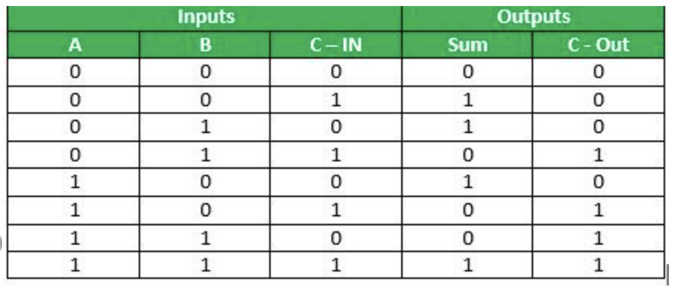
```
      10
     /    \
    5    15
   /  \    /   \
3    7 13 18
```

inorder - 3 5 7 10 13 15 18
preorder - 10 5 3 7 15 13 18
postorder - 3 7 5 13 18 15 10

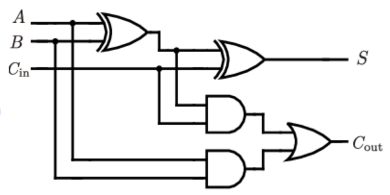
add 21

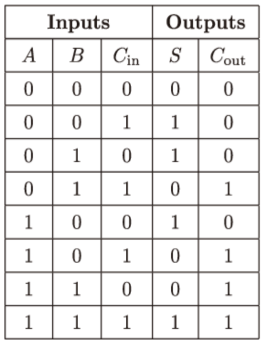
```
      10
     /    \
    5    15
   /  \    /   \
3    7 13 18
           /
          21
```

Full Adder truth table.



Inputs			Outputs	
A	B	C – IN	Sum	C - Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

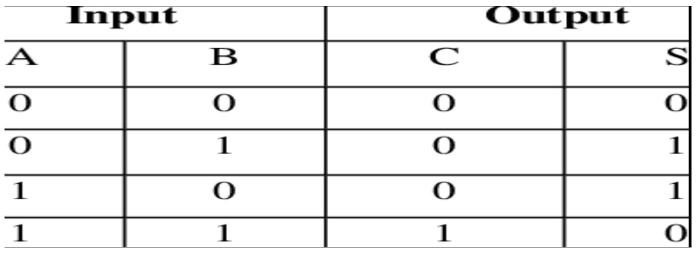




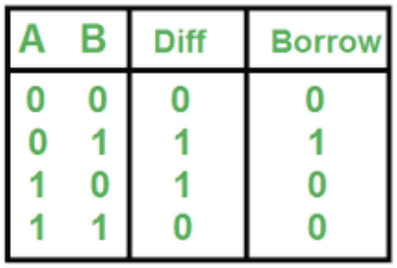
Inputs			Outputs	
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Full-adder circuit diagram and truth table, where A, B, and C in are binary inputs.

Half Adder.



Input		Output	
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



A	B	Diff	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

half substractor