General

- double `*varName` - varName is a double pointer
- Incrementing a pointer increases by the size of variable type
- Every `new` must have a `delete` - to free dynamically allocated mem
- **Pass by value** - func. param and original variable are separated, changes within function not reflected in caller
- **Pass by reference** (`&varName`) - func. parameter is an alias for original variable, changing func. param will modify original variable, `&varName` can't be NULL, can't be reassigned after initialization
- **Pass by pointer** (`*varName`) - takes in mem addr (`&varName`) of original variable, and dereference (`*param`) to access item, changing func. param will modify original variable, `*varName` can be NULL
- **Method Signature** - method name, number of parameters, type of each parameter, order of parameters
- **Function overloading** - same method name, diff method signatures
- **Access modifiers** - `private`, `public`, `protected`

OOP

- **Information Hiding** - `private`/`protected` fields, `public` methods
- **Tell-Don't-Ask** - client should not perform computation

Inheritance

`protected` members can be accessed from inherited classes

```
class Vehicle { public: virtual void honk(); };
class Car: public Vehicle {
    public: virtual void honk(){...}; };
```

Polymorphism

- `virtual` keyword
- **Dynamic binding**
  1. Determine compile-time type of target
  2. Look for all available methods
  3. Choose the most specific method
  4. Determine run-time type of target

Friend

- Other classes can access its `private` & `protected` members and methods
- Not mutual - `List` can access `ListNode` but not the other way
- Not inheritable - children of `List` are not friends of `ListNode`
- Functions (global/local) can be friends as well

```
class ListNode {
    ...
    friend class List;
    friend int functionName(args);
    friend void className::functionName(args);
}
```

Abstract Data Types

Tells what operations can be performed and not reveal implementation

- **Stack** (LIFO) - `push()` - add to top, `pop()` - remove from top
- **Queue** (FIFO) - `enqueue()` - add from back, `dequeue()` - remove from front

Big O notation

Worst case - shows the upper-bound time complexity of program
$T(n) = O(f(n))$ if:

- there exist a constant $c > 0$
- there exist a constant $n_0 > 0$, n_0 = value where $T(n) \leqslant cf(n)$ holds
- such that for all $n > n_0 : T(n) \leqslant cf(n)$

- E.g. if $T(n) = 4n^2 + 16n + 2$, find a c and n_0 where $T(n) \leqslant cf(n)$ holds
- $T(n) < 4n^2 + 24n^2 + 16n^2 = 44n^2 \Rightarrow c = 44, f(n) = n^2, n_0 = 1$
- So, $T(n) = O(n^2)$

- As n tends to infinity, the term with the highest degree will make up most of the value e.g. 99.999% of the result
- **"Naive" way** - $T(n) = O(\text{highest degree})$

- If $T(n) = O(f(n))$ & $S(n) = O(g(n))$, and $10n^2 = O(n^2)$, $5n = O(n)$:
  - $T(n) + S(n) = O(f(n) + g(n))$ - $10n^2 + 5n = O(n^2 + n) = O(n^2)$
  - $T(n) \times S(n) = O(f(n) \times g(n))$ - $10n^2 \times 5n = O(n^2 \times n) = O(n^3)$

- If $T(n) = log_x n$, then $T(n) = O(log_y n)$ holds true for any x, y
  - E.g. $T(n) = log_2 n$, $log_2 n \leqslant c \cdot log_8 n$, $c = log_2 8 = 3$
  - $c \cdot log_8 n = 3 \cdot log_8 n = log_8 n^3$
  - $T(n) = O(log_8 n)$

Common $f(n)$ in Big-O (ascending order of speed)

- Exponential - $O(n!), O(2^n)$
- Quadratic - $O(n^2)$
- Log linear - $O(n \cdot log\ n)$
- Linear - $O(n)$
- Logarithm - $O(log\ n)$
- Constant time - $O(1)$

Algorithm Analysis

- **Loops & Nested loops** - `cost = (# iters) × (max cost of 1 iter)`
- **Sequential statements** - `cost = (cost of 1st) + (cost of 2nd)`
- **If/else** - `cost = max(cost of 1st, cost of 2nd)` $\leqslant$ `(cost of 1st) + (cost of 2nd)`
- **Recursive calls** - $T(n) = 1 + T(n-1) + T(n-2) = O(2^n)$

Divide-and-Conquer - Binary Search in Sorted array

- Find middle element (`mid = (begin+end)/2`)
- If mid element == target, return `true`
- If target > mid element, search on right side (set `begin = 1+mid`)
- Else, search on left side (set `end = mid`)

Time complexity - $O(log(n))$ as keep dividing by 2, array size $= 2^{\#\text{times to cut}}$

Big Ω notation

Best case - shows the lower-bound time complexity of program
$T(n) = \Omega(f(n))$ if:

- there exist a constant $c > 0$
- there exist a constant $n_0 > 0$, n_0 = value where $T(n) \geqslant cf(n)$ holds
- such that for all $n > n_0 : T(n) \geqslant cf(n)$

- E.g. if $T(n) = 4n^2 + 16n + 2$, find a c and n_0 where $T(n) \geqslant cf(n)$ holds
- $T(n) > 4n^2 \Rightarrow c = 4, f(n) = n^2, n_0 = 1$
- So, $T(n) = \Omega(n^2)$

Big Θ notation

Shows the tightest bound time complexity of program
$T(n) = \Theta(f(n))$ if:

- $T(n)$ is $O(f(n))$
- $T(n)$ is $\Omega(f(n))$
- constants c and n_0 don't need to be the same for $O(f(n))$ and $\Omega(f(n))$

Sorting Algorithms

Properties of Sorting Algorithms

- **Stability** - items with same keys stay in same relative positions after sorting
  - e.g. $10_1, 10_2, 30, 20 \rightarrow 10_1, 10_2, 20, 30$
- **In-place sorting algos** - sorting occurs directly within the original memory location, only needing a constant amount of extra mem space

Bubble sort (comparison sort)

- **General idea** - iterate till end from start, if right element > current, swap
- **Invariant** - sorted from the end
- **Big-O**
  - $O(n)$ - finish after 1st loop
  - $O(n^2)$ - total running time $= (n-1) + (n-2) + ... + 1 = \frac{n(n-1)}{2}$
- **Memory** - 1
- stable algo, don't require add. mem space / $O(n^2)$ time complexity (slow for large datasets)

```
for i in range(n):
    swapped = false
    for j in range(0, n-1-i):
        if A[j] > A[j+1]:
            swap(A[j], A[j+1]), swapped = true
    if swapped == false: break
```

Variations - Cocktail sort

- **General idea** - bubble sort forward and backwards
- more efficient than bubble sort / same Big-O as bubble sort, need keep track of start & end indices

Selection sort (comparison sort)

- **General idea** - at a given index, find the smallest element from the right of index and swap
- **Invariant** - sorted from the start
- **Big-O** - $O(n^2)$, $O(n^2)$ - $O(n)$ to check every index in list, $O(n)$ to compare with every element after current index
- **Memory** - 1
- less writes than others, good for small lists / not stable, $O(n^2)$ time

```
for i in range(n-1):
    min_index = i
    for j in range(i+1, n):
        if A[j] < A[min_index], min_index = j
    swap(A[i], A[min_index])
```

Insertion sort (comparison sort)

- **General idea** - LHS is sorted, RHS is not sorted, iterate through list and insert elements from RHS into correct order in LHS
- **Invariant** - sorted within LHS chunk, e.g. items in `A[0] - A[mid]` are sorted
- **Big-O**
  - $O(n)$ - list is already sorted, only 1 iteration
  - $O(n^2)$ - list is randomly ordered or in reverse order

- **Memory** - 1
- stable, efficient for small & nearly sorted lists, space-efficient / inefficient for large lists

```
for i in range(1, n):
    key = A[i], j = i-1
    // reverse bubble sort
    while j >= 0 and key < A[j]:
        A[j+1] = A[j]
        j -= 1
    A[j+1] = key
```

Merge sort (comparison sort)

- **General idea** - divide list recursively into small halves and sort from there
- **Invariant** - items are sorted within 2^x indices
- **Big-O** - $O(n \cdot log\ n)$, $O(n \cdot log\ n)$ - recursive depth is $log\ n$, each level need $O(n)$ to merge
- **Memory** - N, to store levels being sorted currently
- stable, worst-case of $O(n \cdot log\ n)$, good for parallel processing / need add. mem space to store intermediate sorted data, slower than quicksort
- **Note**
  - inside merge sort, insertion sort is used when $n < 1000$
  - stable ver. - in `merge()`, take from left array if elements are identical

```
mergeSort(A, n):
    if (n=1): return
    else:
        left = mergeSort(A[0, n/2], n/2)
        right = mergeSort(A[(n/2)+1, n], n/2)
        return merge(left, right)

merge(left, right):
    result = [], i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i]), i++
        else:
            result.append(right[j]), j++
    result.append(left[i:])
    result.append(right[j:])
    return result
```

Quick sort (comparison sort)

- **General idea** - choose an element x (pivot), rearrange the array around x where LHS is $\leqslant x$ and RHS is $> x$, recursively repeat on LHS & RHS
- **Invariant** - LHS of pivot is $\leqslant$ pivot, RHS of pivot is $>$ pivot
- **Big-O**
  - $O(n \cdot log\ n)$ - pivot divides array into equal halves every time
  - $O(n^2)$ - smallest/largest element is the pivot every time
- **Memory**
  - $O(log\ n)$ - equal halves $\Rightarrow$ $log\ n$ call stack
  - $O(n)$ - unbalanced partitioning $\Rightarrow$ n call stack
- efficient on large datasets, cache friendly as work on same array / not stable, $O(n^2)$ time, not suitable for small datasets

```
quickSort(A, n):
    if (n=1): return
    else:
        p = partition(A, n)
        left = quickSort(A[0, p], p-1)
        right = quickSort(A[p+1,n], n-p)

partition(A, size):
    pivot = 0, packDuplicates(A, size, pivot)
    low = 1, high = size+1
    while low < high:
        while A[low] <= A[pivot] and low < high:
            low++
        while A[high] > A[pivot] and low < high:
            high--
        if low < high: swap(A[low], A[high])
    swap(A[pivot], A[low - 1])
    return low - 1

packDuplicates(A, size, pivotIndex):
    pivot = A[pivotIndex], index = 1
    while index < pivotIndex:
        if A[index] == pivot:
            pivotIndex--
            swap(A[index], A[pivotIndex])
        else: index++
```

Math series

- **Harmonic series** - $\sum_{x=1}^{\infty} \frac{n}{x} = n + \frac{n}{2} + \frac{n}{3} + \cdots = O(n \cdot log\ n)$
- **Geometric series** - $\sum_{x=1}^{\infty} \frac{n}{r^x} = n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots = O(n)$