I SOLID

I.1 Single Responsibility Principle (SRP)

- **Definition**: A class should have only one reason to change, meaning it should only handle one responsibility.
- **Violation**: A `User` class handles both storing user data and sending emails to new users. This means it's responsible for managing data and handling email communication, which are two distinct tasks.
- **Fix**: Create separate classes: a `UserDataService` to handle user data and an `EmailService` to handle email communication.
- **Explanation**: By splitting the responsibilities, we make each class easier to maintain and understand. If there's a change in how we send emails, we only modify the `EmailService`, leaving the `UserDataService` untouched.

I.2 Open/Closed Principle (OCP)

- **Definition**: Classes should be open for extension but closed for modification, allowing new functionality without altering existing code.
- **Violation**: A `DiscountCalculator` class has logic to apply discounts based on customer types like "Regular" and "VIP." When a new customer type, "Premium," is introduced, the class needs modification to add the new discount rule.
- **Fix**: Create individual discount strategies for each customer type that can be applied without modifying the `DiscountCalculator`. For example, use separate classes like `RegularDiscount`, `VIPDiscount`, and `PremiumDiscount` that `DiscountCalculator` can use without needing changes.
- **Explanation**: This makes it easy to add new discounts without modifying existing code, reducing the risk of bugs and keeping the system stable as it grows.

I.3 Liskov Substitution Principle (LSP)

- **Definition**: Subtypes should be substitutable for their base types. If a function uses a base class, it should work with any subclass without issue.
- **Violation**: Suppose there's a `Bird` class with a `fly` method. If we create a `Penguin` class that inherits from `Bird` but overrides the `fly` method to throw an error (because penguins can't fly), it breaks the substitution.
- **Fix**: Use a different hierarchy or interface that better represents different types of birds. For instance, create a `FlyingBird` class that includes `fly` only for birds that actually fly, and a `NonFlyingBird` class for those that don't.
- **Explanation**: This maintains predictable behavior. By ensuring subclasses behave consistently with the base class, the code is easier to understand and maintain.

I.4 Interface Segregation Principle (ISP)

- **Definition**: A class should not be forced to implement interfaces it doesn't use. Instead, create smaller, more specific interfaces.
- **Violation**: Imagine an interface called `Worker` with methods like `startWork`, `stopWork`, and `submitReport`. If some workers only need `startWork` and `stopWork` and have no use for `submitReport`, they're forced to implement an unnecessary method.
- **Fix**: Split the `Worker` interface into smaller interfaces like `Workable` (with `startWork` and `stopWork`) and `Reportable` (with `submitReport`). Classes can implement only the interfaces they need.
- **Explanation**: By dividing interfaces into focused ones, classes only depend on what they actually need. This keeps classes and interfaces lean, reducing complexity and making them easier to maintain.

I.5 Dependency Inversion Principle (DIP)

- **Definition**: High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details.
- **Violation**: A `Report` class directly creates a `PDFGenerator` to generate reports. This tightly couples the `Report` to the specific `PDFGenerator`, making it difficult to switch to another report generator without modifying `Report`.
- **Fix**: Define an interface, such as `ReportGenerator`, that `PDFGenerator` implements. Now, `Report` depends on `ReportGenerator` rather than the concrete `PDFGenerator`, allowing flexibility.
- **Explanation**: By depending on abstractions, the `Report` class can work with any `ReportGenerator` implementation. This makes the system more modular and flexible for changes, as new types of report generators can be added without altering the `Report` class.

II Recursion

- **Tail recursion** uses less memory than non-tail recursion.

II.1 Tail Recursion

II.1.1 Cost of recursion

- Each call to a function adds another frame on the stack
- Each frame contains local variables and parameters and where to return the result

II.1.2 Reducing what needs to be stored

- If we can guarantee we won't need them, we can free the memory for the local variables and parameters.
- We won't need them as long as **we do not use them after the recursive call**.
- Tail recursive functions have the **recursive call as the last thing the function does before it returns**.

Bad: return n x factorial
Good (Tail Recursive): return factorial(n, accumulator)
Three requirements: a base case, a recursive call (the function calls itself) and the function progresses towards the base case

III Complexity

III.1 Master Theorem

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$$T(x) = \begin{cases} \Theta(n^d) \text{ if } a<b^d \\ \Theta(n^d \log(n)) \text{ if } a=b^d \\ \Theta\left(n^{\log_b(n)}\right) \text{ if } a>b^d \end{cases}$$

a: number of subproblems in the recursion
n/b: size of each subproblem
d: the exponent in the cost of the work done outside the recursive calls, specifically in the non-recursive part of the algorithm (like splitting or merging the problem)

e.g. **Binary Search** a=1 b=2 d=0 $\log(n)$ **Merge Sort** a=2 b=2 d=1 $n\log(n)$

III.2 Bound

III.2.1 Big O - Upper Bound

$f(n) = O(g(n))$ iff (if and only if)
$\exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \text{such that } \forall n \geq n_0 : f(n) \leq cg(n)$

III.2.2 Big Ω - Lower Bound

$f(n) = \Omega(g(n))$ iff (if and only if)
$\exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \text{such that } \forall n \geq n_0 : f(n) \geq cg(n)$

III.2.3 Big Θ - Tight Bound

$f(n) = \Theta(g(n))$ iff (if and only if)
$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

III.2.4 Little o upper Bound

$\exists c \in \mathbb{R}^*, \exists n_0 \in \mathbb{N}, \text{such that } \forall n \geq n_0 \text{ such that } f(n) < cg(n)\left(\left|\frac{f(n)}{g(n)}\right| < c\right)$
In other words: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

IV Sorting

IV.1 Selection

repeatedly selects the smallest (or largest) element from the unsorted portion of an array and swaps it with the first unsorted element, moving the sorted boundary one step forward with each iteration

```
for (int i = 0; i < array.size(); i++) {
  // Find min element from i to n-1
  for (int j = i + 1; i < array.size(); j++) {
  }
  // Swap elements at index i and min elements
}
```

IV.2 Insertion

builds the sorted array one element at a time by repeatedly taking the next unsorted element and inserting it into its correct position in the sorted part

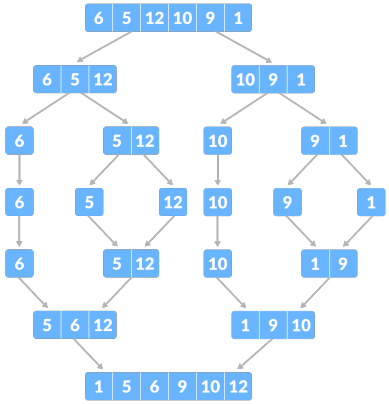
```
for (int i = 1; i < array.size(); i++) {
  for (int j = i; j > 0; j--) {
    if (array.at(j) < array.at(j - 1)) {
      swap(array.at(j - 1), array.at(j));
    } else break;
  }
}
```

IV.3 Bubble

repeatedly compares adjacent elements and swaps them if they are in the wrong order, causing larger elements to "bubble" to the end of the array. With each pass through the list, the largest (or smallest) element settles in its correct position, and this process continues until the entire array is sorted.

```
for (int i = array.size() - 1; i > 0; i--) {
    for (int j = 0; j < i; j++) {
      if (array.at(j) > array.at(j + 1)) {
        swap(array.at(j), array.at(j + 1));
      }
    }
  }
```

IV.4 Merge




```
if (array.size() == 1) return array;
// Sort left and right subarrays
int mid = array.size() / 2;
// ... Define left and right arrays
vector<int> sortedRightArray = sort(rightArray);
vector<int> sortedLeftArray = sort(leftArray);
// Merge left and right subarrays
vector<int> result;
int l = 0, r = 0;
while (l < sortedLeftArray.size() && r <
sortedRightArray.size()) {
  if (sortedLeftArray.at(l) <
sortedRightArray.at(r)) {
    result.push_back(sortedLeftArray.at(l)); l++;
  } else {
    result.push_back(sortedRightArray.at(r)); r+
+;
  }
}
// ... Add remaining elements form
sortedLeftArray or sortedRightArray
return result;
```

IV.5 Quick sort

IV.6 Quick

Place the pivot element at the end of the sub-array (or start, depending on implementation).

Initialize an index i (often called the "store index") to point to the beginning of the sub-array (or just before the first element). This i will keep track of where the next element smaller than the pivot should be placed.

Iterate through the sub-array from the beginning up to (but not including) the pivot's current position.

For each element encountered:

If the current element is less than or equal to the pivot:

Increment i.

Swap the current element with the element at index i. Finally, swap the pivot (which is currently at the end) with the element at i+1. This places the pivot in its final sorted position. The partition function returns the index of the pivot.

```
if (start >= end) return;
// Select the last element as pivot
int pivot = array.at(end);
int pivotIndex = start;
for (int i = start; i < end; i++) {
  if (array.at(i) < pivot) {
    swap(array.at(i), array.at(pivotIndex));
    pivotIndex++;
  }
}
swap(array.at(pivotIndex), pivot);
sort(array, start, pivotIndex - 1);
sort(array, pivotIndex + 1, end);
```

V Linked List & Friend Class

you give access to the class you friend in the original class the ability if an object is instantiated for them to use private and protected members of your class. So in class A you must have 'friend B' or 'friend class B' and then in friend b you will be able to use A stuff.

VI Stack & Queue

VI.1 Stack

- LIFO (Last-in/First-out)
- implemented using linked lists or dynamic arrays
- `push(item)` `pop()` `isEmpty()`
- only have access to the top of the stack

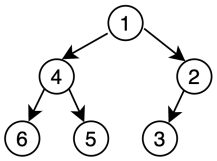
VI.2 Queue

- FIFO (First-in/First-out)
- usually implemented using linked lists
- `enqueue(item)` `dequeue()` `isEmpty()`
- You have access to both the front and back of the queue.

VII Tree

VII.1 Tree Traversals

- Level Order (level by level) 1, 4, 2, 6, 5, 3
- In-order (left, root, right) 6, 4, 5, 1, 3, 2
- Pre-order (root, left, right) 1, 4, 6, 5, 2, 3
- Post-order (left, right, root) 6, 5, 4, 3, 2, 1



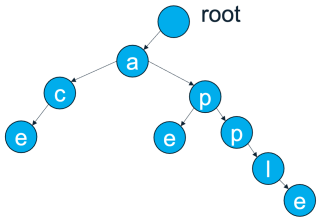
VII.2 BST Delete

- **0 child**: just delete
- **1 child**: set parent's pointer to point to the one child
- **2 children**: replace with in-order successor child (leftmost child of right subtree)

VII.3 BST Insert

- not balanced so easy

VII.4 Trie

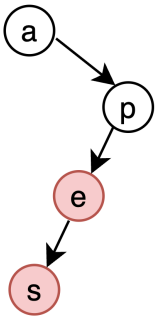


- Tries are a tree data structure that are particularly suited for searching for keys that begin with a specific prefix
- Usually, these keys are strings
- Each path from root represents one key

```
struct TrieNode {
  bool isEndOfWord;
  vector<TrieNode*> childern;
}
```

VII.4.1 Adding keys

- While the key forms a path in the trie, follow the trie while next key character matches the characters in trie
- Add remaining non-matching key characters to tree, mark node with last character as end of word
- Example: Insert "ape" and "apes" - nodes coloured red have `isEndOfWord == true`



VII.4.2 Deleting key

```
// If root is nullptr, key is not in the Trie
if (!root) return nullptr;
// If we've reached the end of the key
if (depth == key.length()) {
  // This node is no longer an end of a word
  if (root->isEndOfWord) root->isEndOfWord =
false;
  // If node has no children, delete it (free
memory) and return nullptr
  if (root->children.empty()) {
    delete root;
    return nullptr;
  }
  return root;
}
// Recursive case for deleting in child
char ch = key[depth];
root->children[ch] = deleteKey(root-
>children[ch], key, depth + 1);
// If no children and not end of word, delete
this node
if (root->children.empty() && !root->isEndOfWord)
{
  delete root;
  return nullptr;
}
return root;
```

To delete a key (word) from a trie, first traverse the trie character by character until you reach the node representing the last letter of the word. At this point, simply set that node's isEndOfWord flag to false. Then, as you backtrack up the trie (often recursively), delete any nodes that are no longer part of any other words (i.e., they are not isEndOfWord for another word and have no other children). This ensures optimal space usage while maintaining the integrity of other words in the trie.

VIII Heap

VIII.1 Binary Tree in array

- If a node is at index i
- Its left child is at $2 * i$
- Its right child is at $2 * i + 1$
- Its parent is at $\lfloor \frac{i}{2} \rfloor$

VIII.2 Operations

Swap the node with the largest(smallest) child until the heap property is satisfied

- **Insert**: bottom to top
- **Delete**: top to bottom
- **Heapify**: bottom to top (back to front)

Heap Insertion (Adding a new node)

Place the new node at the "next available spot": Always add the new element as the next available leaf node to maintain the complete binary tree property. This means you fill levels from left to right.

Graph Visual: The new node just attaches to the tree at the bottom-most, left-most open position.

"Bubble Up" (Heapify-Up): Now, compare the newly added node with its parent.

If the heap property is violated (e.g., in a max-heap, the child is greater than its parent; in a min-heap, the child is smaller than its parent), swap the child and parent.

Graph Visual: The new node "climbs" up the tree, swapping places with its parent, if it's "more extreme" than its parent (larger in max-heap, smaller in min-heap).

Repeat: Keep comparing and swapping with the new node's parent until the heap property is satisfied or the node reaches the root.

Graph Visual: The node keeps moving up level by level until it finds its correct "spot" where it doesn't violate the heap property with its parent. Heap Deletion (Removing the root node, typically)

Note: In a heap, you almost always delete the root (the max element in a max-heap, min element in a min-heap). Deleting arbitrary nodes is more complex.

Replace Root with Last Leaf: Take the very last leaf node in the heap (the rightmost node on the bottommost level) and

move it to the root position. Then, remove the old last leaf node (which is now empty).

Graph Visual: The element at the very bottom-right of your tree "teleports" to the top. The spot it left behind becomes empty.

"Bubble Down" (Heapify-Down): The new root might violate the heap property. Compare it with its children.

In a max-heap: If the new root is smaller than either child, swap it with the larger of its two children.

In a min-heap: If the new root is larger than either child, swap it with the smaller of its two children.

Graph Visual: The new root "sinks" down the tree, swapping places with its "most extreme" child (largest in max-heap, smallest in min-heap) if it violates the heap property.

Repeat: Continue comparing and swapping with the new node's children until the heap property is satisfied (it's "more extreme" than its children, or it becomes a leaf node).

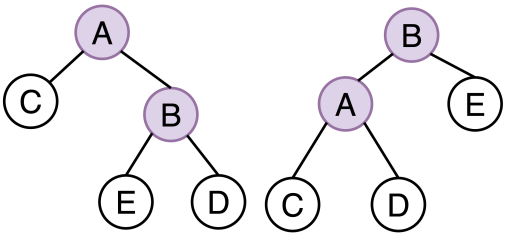
Graph Visual: The node keeps moving down level by level until it finds its correct "spot" where it doesn't violate the heap property with its children.

IX Red Black Tree

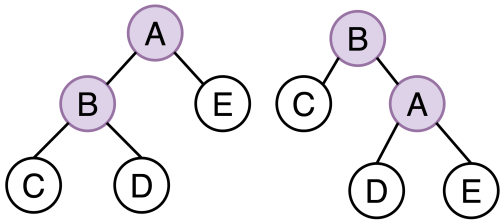
1. Root is always black
2. No two adjacent 临近的 nodes are red
3. Any path between a node and any descendant (lower) node 子孙 has the same number of black nodes

IX.1 Rotate

Left



Right



IX.2 Insert

IX.2.1 Root

Change colour to black

IX.2.2 Violated 2 & Uncle is red

- Change parent and uncle to black
- Change grandparent to red
- Make grandparent n, and repeat

IX.2.3 Violated 2 & Uncle is black

IX.2.3.1 Left Left

rotate right, swap colours of parent and grandparent

IX.2.3.2 Left Right

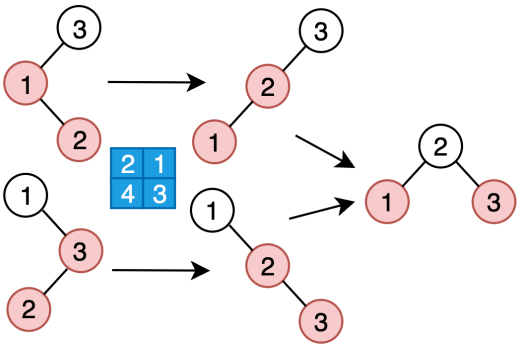
rotate left, then right, swap colours of new node and grandparent

IX.2.3.3 Right Right

rotate left, swap colours of parent and grandparent

IX.2.3.4 Right Left

rotate right, then left, swap colours of new node and grandparent



IX.3 Delete

IX.3.1 Simple Cases

- If a node is red with nullptr child (no children)
- If a node has 1 child and **either** the node OR child (but not both) is red

1. Delete node 2. Updated node → **black** (node replaced the deleted node)

IX.3.2 Double Black

If both node to be deleted AND child are **black** (or the node has no children), the updated node becomes **double black**

- n's sibling is **black**
  - with at least one **red** child
    1. Rotate (as per insertion following path to red child)
    2. Recolour red child to **black**, sibling to red
  - with all children **black**
    1. Recolour sibling to red
    2. Push **black** up (**black** parent → **double black**, red parent → **black**)
- n's sibling is **red**
  1. Rotate
  2. Recolour Sibling to **black** Parent to red

X Selecting Data Structures and Algorithmic Strategies Quiz

X.1

- **Heap Sort**: Transform and Conquer
- **Memoisation**: Synamic Programming
- **Selection Sort**: Brute Force
- **Quick Sort**: Divide and Conquer

X.2

You need to store a list of words and know that words that were recently searched for are likely to be searched for again. **ANSWER: Linked List**

$O(n)$ to find a word and when a word is found, move it to the head (most recent will be at front of list) $O(1)$. The most recently searched for words will always be at the start of the list, improving the average case to find these words $O(1)$ if most recently searched word.

X.3

- **Linear Structures**: Queue, Linked List, Array, Heap, Stack, Vector
- **Non-linear Structures**: Red-Black Tree, Binary Search Tree, Trie

X.4

Which of the following C++ containers could be shuffled using the C++ shuffle function from the algorithm library?
ANSWER: vector, array

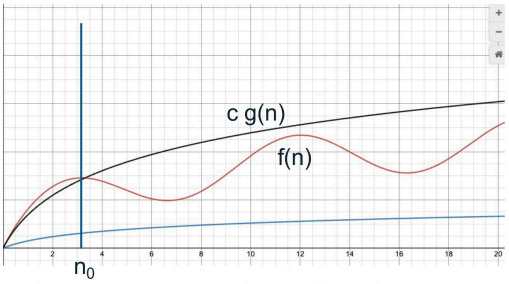
X.5

- `list` : Doubly Linked List
- `priority_queue` : Heap
- `map` `set` : Red-Black Tree (balanced binary search tree)

Big O – Upper bound

We say that $f(n) = O(g(n))$ iff (if and only if)

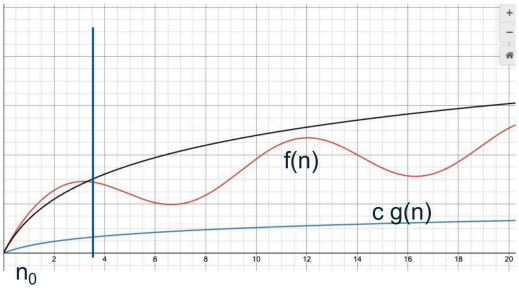
$\exists c \in R^+, \exists n_0 \in N,\ such\ that\ \forall n \geq n_0 : f(n) \leq cg(n).$



Big Ω– Lower bound

We say that $f(n)$ is in $\Omega(g(n))$ iff (if and only if)

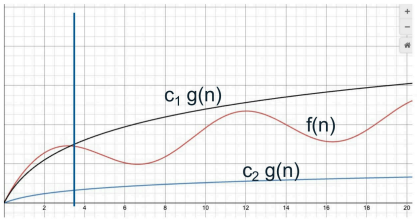
$\exists c \in R^+, \exists n_0 \in N,\ such\ that\ \forall n \geq n_0 : f(n) \geq cg(n).$



Big Θ Tight bound

$f(n)$ =Θ$(g(n))$ iff (if and only if):

$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.



Polynomials of degree k are in $\Theta(n^k)$:
$a_k n^k \leq a_k n^k + \ldots + a_1 n + a_0 \leq (a_k + \ldots + a_1 + a_0) n^k$

Little o Upper bound

$\forall c \in R^+, \exists n_0 \in N, such\ that\ \forall n \geq n_0\ such\ that\ f(n) < cg(n)\ (\left|\frac{f(n)}{g(n)}\right| < c).$

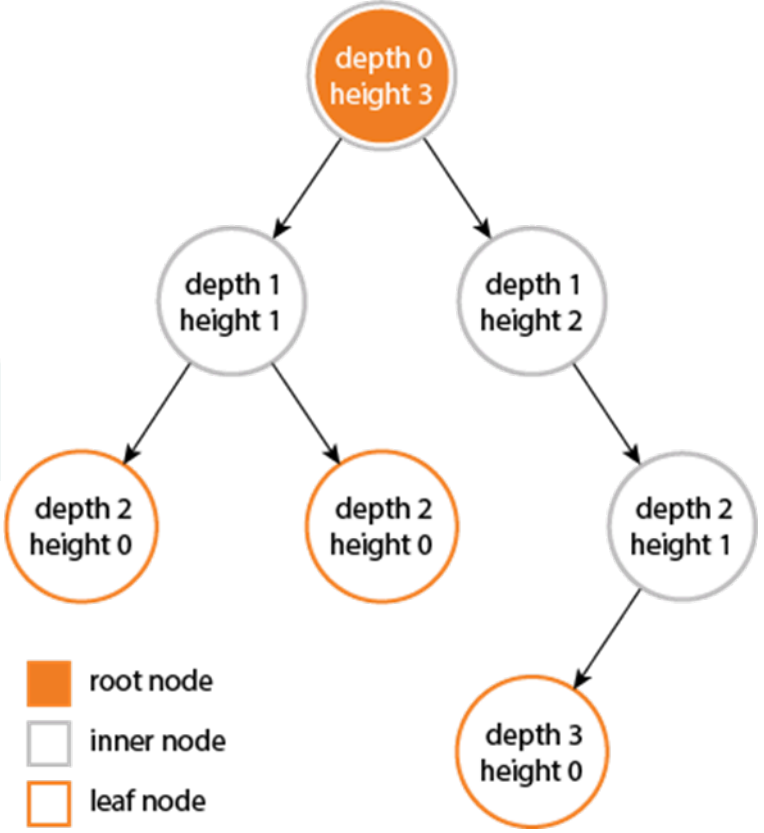
In other words:
$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$

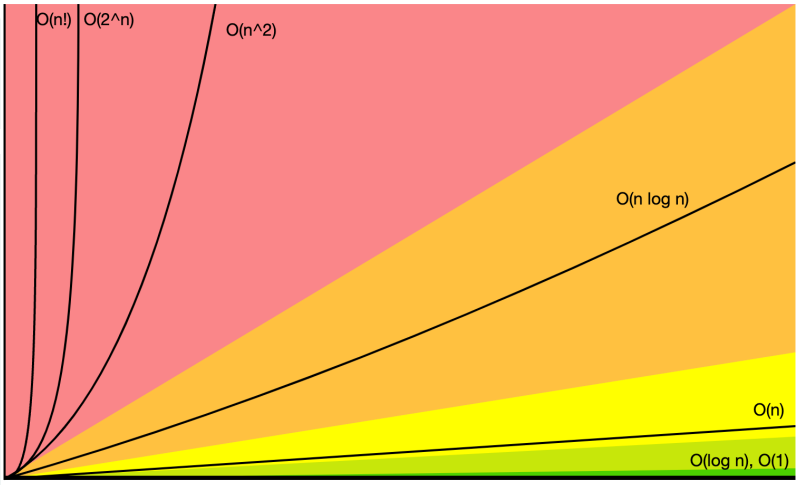
Examples:

If f(n)=o(g(n)) then f(n)=O(g(n)).

$n = o(n^2)$.
$\log n = o(n)$.

However, Little o does not allow the same growth rate.





	Best	Average	Worst	Space	Stable
Selection	$O(n^2)$			$O(1)$	
Insertion	$O(n)$	$O(n^2)$		$O(1)$	✓
Bubble	$O(n^2)$			$O(1)$	✓
Merge	$O(n \log(n))$			$O(n)$	✓
Quick	$O(n \log(n))$		$O(n^2)$	$O(\log(n))$	
Bucket	$O(n)$	$O(n+k)$ (k: numbers of buckets)	$O(n^2)$	$O(n+k)$	✓
Counting	$O(n+k)$ (k: range of the input values, max-min)			$O(n+k)$	✓
Heap	$O(n \log(n))$			$O(1)$	

	Worst			Average			Space
	Insert	Delete	Search	Insert	Delete	Search	
Vector Ordered	$O(n)$		$O(\log(n))$	$O(n)$		$O(\log(n))$	$O(n)$
Vector Unordered	$O(1)^*$	$O(n)$		$O(1)^*$	$O(n)$		
Linked List	$O(1)$			$O(1)$			
Binary Search Tree	$O(n)$			$O(\log(n))$			
Balanced BST (RBT)	$O(\log(n))$			$O(\log(n))$			
Priority Queues	**Insert**	**RMH****	**Peek**	**Insert**	**RMH****	**Peek**	
Linked List	$O(n)$	$O(1)$		$O(n)$	$O(1)$		
Heap	$O(\log(n))$		$O(1)$	$O(1)$	$O(\log(n))$	$O(1)$	

*: Amortised - ie over a sequence of this operation. Resizing vector $O(n)$
**: Remove highest priority

X.6

We are implementing a scheduler for choosing the next process to run on a computer. System processes, that keep the computer running are more important than user processes (someone's C++ homework); but otherwise processes should be run on a first come first served basis Assuming we decide to keep all of these processes in a single ordered array, which sorting algorithms could be used (assume the base algorithms as you learned in this course, not variations). **ANSWER: Bubble Sort, Insertion Sort, Merge Sort**