# CS2040S Cheatsheet
Xu An Teh

| Pre-empts | |
|---|---|
| Restrictions for CS2040S | Avoid Java "advanced" features like Lambda expressions, Type inferences (var), Default, static private methods in an interface...<br>Do not use libraries if they make the problem set easier<br>Do not use libraries unless the problem set specifically says you can |
| Considerations | Only makes code shorter<br>Little extra functionality<br>Often hide what is really happening<br>May or may not make code harder / easier to read |
| Goal of writing code in CS2040S | Correct / Bug-Free<br>Easy to read / understand<br>Efficient<br>Submitted by deadline<br>Short |
| Advice | Make your code intentional (Avoid default / non-explicit behaviours) |

| OOP Paradigm (Same concept as CS2030S, different applications) | |
|---|---|
| Abstraction | User-centric, information on a need-to-know basis, hide implementations where possible |
| Encapsulation | Group methods and data in a class meant to represent something (noun), Hiding implementation and only interface publicly visible. |
| Inheritance | Build new classes by extending existing classes (Sharing and Adding functionality) |
| Polymorphism | Same interface, but different behaviour based on context |

| Application of OOP Paradigm (for Algorithmic Design) | |
|---|---|
| Application | Divide Problem into Components<br>Define interface between components<br>Solve each problem separately<br>Repeat, then combine solutions |
| Abstraction | Interface: how you manipulate the object<br>Implementation: details hidden inside the object |
| Encapsulation | Class as a template for producing an object<br>Grouping functionalities to solve a subset of problems |

| Principles of Java | |
|---|---|
| First Principle | Everything is an Object |
| Second Principle | Everything has a Type |

| Classes & Objects, Regular & Static | |
|---|---|
| Classes | Template for how to make an object |
| Objects | An instance of the class |
| Constructors | Creates and instantiate the object and its fields |
| Parts of an Object | State (data), Behaviour (methods for modifying the state) |

| | |
|---|---|
| Regular vs Static | Regular Variables/Functions are PER OBJECT<br>Static Variables/Functions are PER CLASS |

| Access Control | |
|---|---|
| (none specified) | Within the same package |
| public | Everywhere |
| private | Only in the same class |
| protected | Within the same package, and by subclasses |
| Advice: Always specify the access you intend (even if the default is okay) | |

| Java Operators | |
|---|---|
| = | Assignment |
| +, -, *, / | Plus, minus, multiplication, division |
| % | Remainder, Modulo |
| ++, -- | Increment, decrement |
| <, > | Less than, greater than |
| <=, >= | Less-than-or-equal, greater-than-or-equal |
| &&, \|\| | Logical and, logical or |
| ~, &, ^, \| | Bitwise operations: complement, and, xor, or |

| Primitive Data Types | | | |
|---|---|---|---|
| Byte | 8 bit | -2^7 | 2^7 - 1 |
| Short | 16 bit | -2^15 | 2^15 – 1 |
| Int | 32 bit | -2^31 | 2^31 - 1 |
| Long | 64 bit | -2^63 | 2^63 - 1 |
| Float | 32 bit (IEEE 754) | (2 – 2^23) * | |
| Double | 64 bit (IEEE 754) | +-(2 ^ -1074) | +-((2 - 2^-52) * 2^1023) |
| Boolean | 1 bit | False | True |
| Char | 16 bit (Unicode) | \u0000 (0) | \uffff(65535) |

| Algorithm Analysis (Big O notation) | |
|---|---|
| Pre-empt | Take Logs to be Base 2, Loga(n) = Log2(n)/log2(a)<br>Always think big inputs |
| Big-O Notation $T(n) = O(f(n))$ | $\exists c > 0 \land \exists n_0 > 0 \rightarrow \left(\forall n > n_0\left(T(n) \le cf(n)\right)\right)$ |
| Big-Ω Notation $T(n) = \Omega(f(n))$ | $\exists c > 0 \land \exists n_0 > 0 \rightarrow \left(\forall n > n_0\left(T(n) \ge cf(n)\right)\right)$ |
| Big-Θ Notation $T(n) = \Theta(f(n))$ | $T(n) = O(f(n)) \land T(n) = \Omega(f(n))$ |
| Order of Size: | (see table below) |
| Summation | $T(n) = O(f(n)) \land S(n) = O(f(n))$<br>$\rightarrow T(n) + S(n) = O(f(n) + g(n))$ |
| Product | $T(n) = O(f(n)) \land S(n) = O(f(n))$<br>$\rightarrow T(n) * S(n) = O(f(n) * g(n))$ |

Order of Size:

| Function | Name | Function | Name |
|---|---|---|---|
| 5 | Constant | $n^3$ | Polynomial |
| loglog(n) | Double Log | $n^3$log(n) | |
| log(n) | Logarithmic | $n^4$ | Polynomial |
| $\log^2(n)$ | Polylogarithmic | $2^n$ | Exponential |
| n | Linear | $2^{2n}$ | |
| nlog(n) | Log-linear | n! | Factorial |

| | |
|---|---|
| Sterling's Approximation | $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ |
| Sequential Statements | $cost = cost_{first} + cost_{second}$ |
| If / else statements | $cost = max(cost_{first}, cost_{second}) \le cost_{first} + cost_{second}$ |
| Recursion (Summations) | Geometric Sum: $\sum_{k=0}^{n} ar^k = \begin{cases} a(n+1) & a = 1 \\ a\left(\frac{1-r^{n+1}}{1-r}\right) & otherwise \end{cases}$<br>Arithmetic Sum: $\sum_{i=1}^{n}(a + d * i) = \frac{n(2*a+(n-1)d)}{2}$<br>Master Theorem |

| Searching Algorithms | |
|---|---|
| Characteristics | Runtime |
| Search Algorithm | Linear Search: O(n); Binary Search: O(log(n))<br>Quick Search: O(log(n)) |
| Linear Search | Check all elements |
| Binary Search | Check mid element, compare with required value, Must be searching an ordered array |
| Quick Search | Check Relative Positioned element, compare with required value, Must be searching an ordered array |
| Precondition | Fact that is true when the function begins<br>Something important for it to work correctly<br>Useful to validate when possible |
| Postcondition | Fact that is true when the function ends<br>Something useful to show that the computation was done correctly |
| Invariant | Relationship between variables that is always true |
| Loop Invariant | Relationship between variables that is true at the beginning (or end) of each iteration of a loop |
| Peak Finding | Find Local Max, Binary Search, Check slope direction<br>Invariants: There is a peak in the range [begin, end]<br>Every peak in [begin, end] is a peak in [0, n-1] |
| Steep Peaks | Steep peaks are strictly larger than its neighbours |
| 2D Peak | 2D Peak: Larger than or equal to its neighbours<br>Find Max of All Col, then find peak (O(mn + log(m)))<br>Find Local of All Col, then find peak (incorrect)<br>Find Max of Mid Col, then recurse (O(nlog(m))<br>Find Max of Border + Cross, then recurse in quadrant where (the neighbour of Max) > Max (O(n+m)) |

| Sorting Algorithms | |
|---|---|
| Characteristics | Runtime, Space, Stability, Worst Cases |
| Runtime | Best Case (Ω), Average Case (Θ), Worst Case (O, Impt) |
| Space | Total Space ever allocated<br>(Alt, Realistic: Max Space allocated at one time) |
| Stability | Preserves order of equal elements |
| Bogo Sort | Randomly permutate array, check if sorted, O(n*n!), Unstable, All cases are worst cases |

| | |
|---|---|
| Quantum Bogo Sort | Generate permutation and check the array if sorted, destroy universe if not, If many-worlds interpretation holds, there exists a surviving universe where array is sorted, O(n) |
| Bubble Sort | Iterate through the array, swap if greater than next element, loop first n-1 element<br>Invariant: Largest k element sorted at k loops<br>$\Omega(n)/O(n^2)$, In-Place, Stable, Reversed / Circular Left Shift |
| Selection Sort | Iterate through the array, swap minimal element to the front, loop last n – 1 element<br>Invariant: smallest k element sorted at k loops<br>$\Omega(n)/O(n^2)$, In-Place, Unstable, All cases are worst cases |
| Insertion Sort | Take first element, swap insert into sorted array at the front, loop to next unsorted element<br>Invariant: smallest k element sorted at k loops<br>$\Omega(n)/O(n^2)$, In-Place, Stable, Reversed |
| Merge Sort | Split array in half, recurse halves, merge in order<br>Invariant: Subarrays are sorted at end of loop<br>$\Theta(n\log(n))$, Space: $\Omega(n)/O(n^2)$ by implementation,<br>Stable (check merge), All cases are worst case |
| Ingrassia-Kurtz Sort | Generate all permutations, sort permutations, return first element in the sorted list of permutations |
| Quick Sort | Partition the array on pivot by swapping bigger elements on the left with smaller elements on the right, then recurse<br>Invariant (Partition): for every i < low, A[i] < pivot,<br>for every j > high, A[j] > pivot<br>Runtime dependent on pivot selection, In-Place, Unstable, All cases are worst case<br>Runtime: $1^{st}$ elem = $\Omega(n^2)$, Median elem = $O(n\log(n))$,<br>1/10+9/10 = $O(n\log(n))$ |
| Quick Sort (Duplicate) | 3-Way Partitioning:<br>1) Two Pass: Regular Partition then Pack Duplicates<br>2) One Pass: More Complex, Maintain four regions of array<br><pivot, =pivot, in-progress, >pivot (4 pointers) |
| Paranoid Quick Sort | Randomise pivot index selection<br>$\Theta(n\log(n))$ Runtime |

| Data Structure Design | |
|---|---|
| Data Structure | A way of storing and organizing data efficiently, such that required operations can be performed efficiently with respect to time as well as memory<br>Considerations: Maintenance, Modification, Query<br>Upgrades: Augmentations, New Properties |
| Static Data Structure | Size of Structure is fixed; Content can be modified but without changing memory space allocated to it<br>Eg. Array, Stack, Queue, Fixed Size Tree |
| Dynamic Data Structure | Size of Structure is not fixed and can be modified during the operations performed on it<br>Eg. Lists, Trees, Tries, Hash Tables |
| Augmenting Data Structures | 1) Choose underlying data structure<br>2) Determine additional info needed<br>3) Modify data structure to maintain additional info when structure changes |

| | |
|---|---|
| | 4) Develop new operations |
| Order Statistics | Preprocessing, Accessing, Modifying, Postprocessing |

| Tree Data Structure | |
|---|---|
| Idea | Given a dictionary, storing key-value pairs<br>Possible Implementations<br>Sorted Array } insert: O(n), (binary) search: O(log(n))<br>Unsorted Array } insert: O(1), search = O(n)<br>Linked List } insert: O(1), search = O(n)<br>Balanced BS Trees } insert: O(log(n)), search = O(log(n)) |
| Trees | Components: Nodes (1 Root), Edges, No Cycles |
| Binary Trees | Empty or A node pointing to two binary trees |
| BST | Keys in left sub-tree < key < Keys in right sub-tree |
| Root | The base node, all search/insert start here |
| Leaf | No children, Height = 0, Weight = 1 |
| Siblings | Nodes that share a parent |
| Height | -1 if null, 0 if leaf, else max(h(v.left), h(v.right)) + 1 |
| Weight | 0 if null, 1 if leaf, else w(v.left) + w(v.right) + 1 |
| Rank | r(leftparent) + r(v.left) |
| Shape | Same keys != Same Shape, affects performance, determined by order of insertion of nodes<br># orders: n!; # shapes: $\sim 4^n$ (Catalan) |
| Tree Traversal | Pre-Order, In-Order, Post-Order, Level-Order<br>Order of visited nodes |

| Binary Search Tree (BST) | |
|---|---|
| Description | Keys in left sub-tree < key < Keys in right sub-tree |
| Applications | Max/min, rank/select, successor/predecessor operations |
| Search | At each node, compare node key, go to key direction |
| Insert | Search, then add at null |
| Delete | No child: remove v<br>1 child: remove v, connect child(v) to parent(v)<br>2 child: x = successor (v), delete(x), remove v, connect x to left(v), right(v), parent(v) |
| Successor / Predecessor | Successor: Get right child left most node, else left parent<br>Predecessor: Get left child right most node, else right parent |
| Runtime Summary | Insert, delete, search, predecessor, successor, findMax, findMin: O(h); in-order-traversal: O(n) |
| Balanced | h = O(log(n)), for Balanced BST: all operations are O(log(n)) |
| Getting a Balanced Tree | 1) Define good property of a tree<br>2) Show that if the good property holds, then the tree is balanced<br>3, Invariant) After every insert/delete, make sure the good property still holds, If not, fix it |
| AVL Tree | Adelson-Velskii & Landis 1962 Tree<br>Step 0, Augment: every node v, store height<br>Update on insert/delete operations<br>Step 1, Define Height Balance: node v is height-balanced if $\lvert v.left.height - v.right.height \rvert \leq 1$ |

| | |
|---|---|
| | Binary Search Tree is height balanced if every node in the tree is height balanced / # keys in heavier sub-tree at most twice of # keys in lighter sub-tree<br>Step 2, Maintain Height Balance: Tree Rotation |
| Claim | A height-balanced tree with n nodes has at most height h = O(log(n)) |
| Tree Rebalancing: Tree Rotation | Right Rotation on Node v: v.left = vLeft.right, vLeft.right = v<br>Left Rotation on Node v: v.right = vRight.left, vRight.left = v<br>Maintains ordering of keys => Maintains BST Property |
| LR/RL-Heavy | Left-Right-Heavy: Left Rotate Left Child, then Right Rotate<br>Right-Left-Heavy: Right Rotate Right Child, then Left Rotate |
| Insert in AVL | Insert key in BST, then walk up tree and check for balance<br>Only need to fix lowest out-of-balance node<br>Only at most 2 rotations to fix |
| Delete in AVL | If v has 2 children, swap with successor<br>Delete node v and reconnect children<br>Check every ancestor of deleted node for height-balance<br>At most O(log(n)) rotations to fix |

| Trie | |
|---|---|
| Description | Trees where nodes can have many children<br>Used for storing address and words (ie Dictionary) |
| Root-to-Leaf Path | Represent Strings ie Keys |
| Terminating Character | Marks the end of String ie Keys |
| Space Required | O((size of text)*overhead) |
| Search | O(L) where L is length of string |
| VS Trees | Shorter Runtime, Bigger Space, No Ordering |
| Trie Nodes | Many Children, for Strings: Fixed degree (ASCII: 256) |