# Section 1 Big O

Name of complexity [Asymptomatic analysis] in order

* Constant Time: O(1)
* Logarithmic Time: O(logn) binary search
* Linear Time: O(n)
* Linearithmic Time: O(nlogn) fastest sorting alg.
* Quadratic Time: O(n2) nested for loops in brute force
* Exponential Time: O(2n) recursion
* Factorial Time: O(n!) rare

Rule to calculate O

1. Consider only worst case [even if we have break in a loop]
2. Remove constant [O(4n+1)]
3. Different terms for inputs should consider separated [O(n+m), O(n\*m)]
4. Drop non-dominant terms [follow the complexity chart]

Spacing complexity

* Don’t care about input, only care about the spacing we create

# Section 2 Data structure: Array – [1,2,3,4….]

Time complexity of array operations

* Lookup: O(1)
* Append: O(1)
* Insert: O(n) [iteratively shift index]
* Delete: O(n) [iteratively shift index]

Static and dynamic array

* The space of static array is fixed in memory
* Change size of dynamic array: claim new space -> copy old array and new items to a new space. This cost is O(n) for a simple operation such as append in the very low-level. However, it does not matter to the calculation of your code when using such as array.push()

Pro and Con use Array

* Pros: fast lookup, fast push/pop, ordered
* Cons: slow insert, slow delete, fixed array fixed size

# Section 3 Data structure: Hash Table – {a: 1, b:2, …}

Time complexity of hash table operations

* Lookup: O(1)
* Append: O(1)
* Insert: O(1)
* Delete: O(1)

Truth of hash table

* Key -> hash function (really fast depends on language) -> memory address space -> value
* Hash collision: same hash values collision in a memory space. Several methods to resolve. May cause the performance down to O(n)
* Commonly use hash table to optimize nested loop to decrease O(n^2) to O(n)

# Section 4 Data structure: Linked Lists

Time complexity of linked lists

* Lookup: O(n)
* Append: O(1)
* Prepend: O(1)
* Insert: O(n)
* Delete: O(n)

Concept

* Node contains [content, pointer], where pointer points to the next node
* Linked list has a head and tail, where the tail’s pointer points to null
* Sorted data structure
* If lost head, the list will be garbage collected

Doubly linked lists

* One more pointer that points to the previous item
* More efficient in searching and inserting O(n/2), but with more memory space

!!

* Linked lists properties help build up Queues, Stacks, and Trees
* Linked lists solve Hash value collision

# Section 5 Data structure: Stacks and Queues

## Stacks - LIFO

* Supported operations: lookup O(n), pop O(1), push O(1), peek O(1)
* Applications: function calls, browser history, undo…
* Construct stack by array because popping and pushing item come with O(1)

## Queues - FIFO

* Supported operations: lookup O(n), enqueue O(1), dequeue O(1), peek O(1)
* Applications: wait list…
* Construct queue by linked list, because adding head or tail come with O(1)

# Section 6 Data structure: Tree

## Tree structure helps search faster O(log N)

## Perfect Binary Tree

* All leaf nodes are filled
* The number of nodes at each level double [level n has 2n nodes]
* The number of nodes at a level equals to the sum of all nodes in upper levels

## Binary Search Tree

* Maintain an order where left child less than current node, right child greater than current node
* Complexity
  + Lookup: O(log N)
  + Insert: O(log N)
  + Delete: O(log N)
* Unbalanced BST may cause complexity down to O(n)
* AVL tree and Red/Black tree are self-balancing trees upon insertion

## Binary Heap

* Only require that child values are less than their parent
* Used for priority queue
* Easy to find max or max group
* Insertion follows left to right from bottom level, they bubble up if needed

# Section 7 Data structure: Graph

Types of graphs

* Directed, undirected
* Weighted, unweighted
* Cyclic, acyclic

# Section 8 Algorithms: Recursion

* A recursive function needs a base case (stopping condition)
* In function calling stack, the most internal return should be bubbled up returned to the top
* Criterion of writing a recursion
  + Have a base case
  + Call itself within the function
  + Two returns, one within base case, one return itself
* Recursion might have O(2n) without optimization
* Anything you do with recursion can be done by iteration
  + Recursion keeps code simple and readable
  + It requires large function calling stack
  + Best case: when you don’t know how deep your data structure is
    - Traversing through trees or graphs
    - Some sorting algorithms
    - Divide and conquer

# Section 9 Algorithms: Sorting

Complexity

* Bubble sort: O(n) O(n2) O(n2) O(1)
* Selection Sort: O(n2) O(n2) O(n2) O(1)
* Insertion Sort: O(n) O(n2) O(n2) O(1)
* Merge Sort: O(nlogn) O(n)
* Quick Sort: O(nlogn) O(n2) O(logn)

Pros & Cons

* Bubble sort / selection sort: never use, only for introduction
* Insertion sort: input is small, or items are nearly sorted use
* Merge sort: divide and conquer, fast, expensive in space
* Quick sort: mostly used, better than merge sort, but worst case if not picking pivot properly

Concept

* Mathematically, sorting algorithms cannot beat O(nlogn) since we are comparing
* There are some non-comparison sorting algorithms work with fixed length integers
  + Radix sort: O(nk) O(nk) O(nk) O(n+k)
  + Counting sort O(n+k) O(n+k) O(n+k) O(n)

# Section 10 Algorithms: Search & Traversal

Search Types

* Linear search: O(n) loop over all to search
* Binary search: O(logn) if data is sorted, divide and conquer

Traversal Types for graphs and trees

* Breadth First Search: O(n)
* Depth First Search: O(n)

Pros and Cons for traversal

* Time complexity for BFS and DFS are the same
* BFS
  + Good to find the shortest path between starting and other nodes
  + Requires more memory to keep the track
  + If target node is in higher level of tree, use BFS
* DFS
  + Solve the problem that finds if a node exists
  + Less memory use, but may get slower if tree or graph is deep
  + If target node is in lower level of tree, use DFS

Three orders in DFS

9

4 20

1 6 15 170

* In-order: the result is sorted in BST [1, 4, 6, 9, 15, 20, 170]
* Pre-order: easy to reconstruct tree [9, 4, 1, 6, 20, 15, 170]
* Post-order: [1, 6, 4, 15, 170, 20, 9]

Extra algorithms for finding shortest path with weights in a graph

* Bellman-Ford: worst in O(n2)
* Dijkstra: more efficient, but no negative weight

# Section 11 Algorithms: Dynamic Programming

* It is an optimization technique using cache (Memorization)
* Dynamic Programming: Divide & Conquer + Memorization
* Closure is one important concept in using memorization

Mind flow

* A problem can be divided into subproblems
* We have a recursive solution
* Find the repetitive subproblems then memorize the results of subproblems