# Tech Interview Cheat Sheet

This list is meant to be both a quick guide and reference for further research into these topics. It's basically a summary of that comp sci course you never took or forgot about, so there's no way it can cover everything in depth.

## Contributing

This is an open source, community project, and I am grateful for all the help I can get. If you find a mistake make a PR and please have a source so I can confirm the correction. If you have any suggestions feel free to open an issue.

## Challenges

This project now has actual code challenges! This challenges are meant to cover the topics you'll read below. Maybe you'll see them in an interview and maybe you won't. Either way you'll probably learn something new. Click here for more

# Table of Content


- Asymptotic Notation
- Data Structures
  - Array
  - Linked List
  - Hash Table or Hash Map
  - Binary Tree
- Algorithms
  - Algorithm Basics
  - Search Algorithms
    - Breadth First Search
    - Depth First Search
  - Sorting Algorithms
    - Selection Sort
    - Insertion Sort
    - Merge Sort

        - Quick Sort
- Additional Resources


# Asymptotic Notation

## Definition:

Asymptotic Notation is the hardware independent notation used to tell the time and space complexity of an algorithm. Meaning it's a standardized way of measuring how much memory an algorithm uses or how long it runs for given an input.

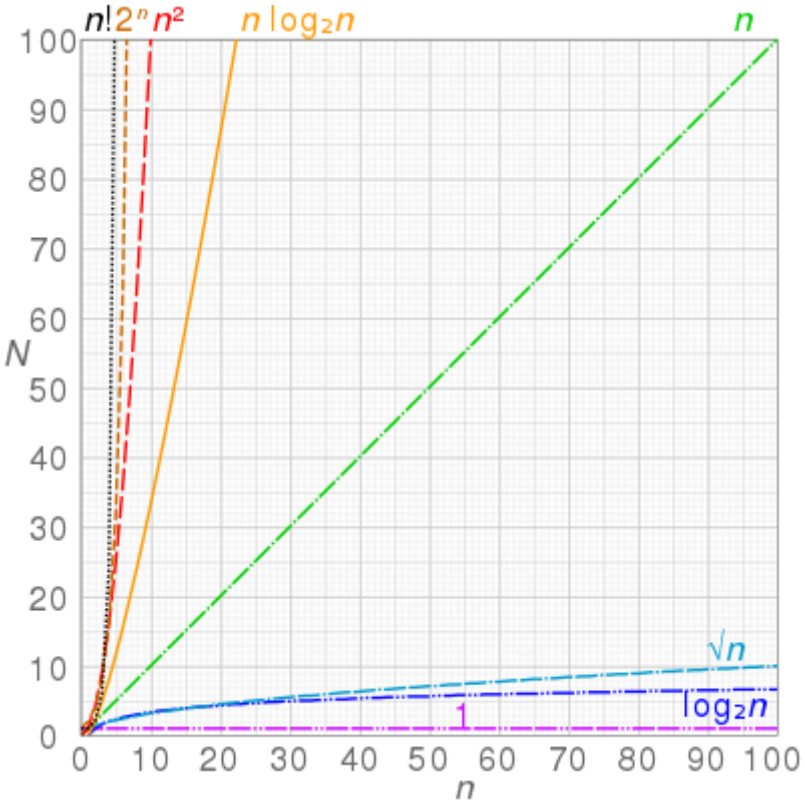
### Complexities

The following are the Asymptotic rates of growth from best to worst:

- constant growth - `O(1)` Runtime is constant and does not grow with `n`
- logarithmic growth – `O(log n)` Runtime grows logarithmically in proportion to `n`
- linear growth – `O(n)` Runtime grows directly in proportion to `n`
- superlinear growth – `O(n log n)` Runtime grows in proportion *and* logarithmically to `n`
- polynomial growth – `O(n^c)` Runtime grows quicker than previous all based on `n`
- exponential growth – `O(c^n)` Runtime grows even faster than polynomial growth based on `n`
- factorial growth – `O(n!)` Runtime grows the fastest and becomes quickly unusable for even small values of `n`

(source: Soumyadeep Debnath, *Analysis of Algorithms | Big-O analysis*)

Visualized below; the x-axis representing input size and the y-axis representing complexity:




(source: Wikipedia, *Computational Complexity of Mathematical Operations*)

### Big-O notation

Big-O refers to the upper bound of time or space complexity of an algorithm, meaning it worst case runtime scenario. An easy way to think of it is that runtime could be better than Big-O but it will never be worse.

### Big-Ω (Big-Omega) notation

Big-Omega refers to the lower bound of time or space complexity of an algorithm, meaning it is the best runtime scenario. Or runtime could worse than Big-Omega, but it will never be better.

### Big-θ (Big-Theta) notation

Big-Theta refers to the tight bound of time or space complexity of an algorithm. Another way to think of it is the intersection of Big-O and Big-Omega, or more simply runtime is guaranteed to be a given complexity, such as `n log n`.

### What you need to know

- Big-O and Big-Theta are the most common and helpful notations
- Big-O does *not* mean Worst Case Scenario, Big-Theta does *not* mean average case, and Big-Omega does *not* mean Best Case Scenario. They only connote the algorithm's performance for a particular scenario, and all three can be used for any scenario.

- Worst Case means given an unideal input, Average Case means given a typical input, Best case means a ideal input. Ex. Worst case means given an input the algorithm performs particularly bad, or best case an already sorted array for a sorting algorithm.
- Best Case and Big Omega are generally not helpful since Best Cases are rare in the real world and lower bound might be very different than an upper bound.
- Big-O isn't everything. On paper merge sort is faster than quick sort, but in practice quick sort is superior.

# Data Structures

## Array

### Definition

- Stores data elements based on an sequential, most commonly 0 based, index.
- Based on tuples from set theory.
- They are one of the oldest, most commonly used data structures.

### What you need to know

- Optimal for indexing; bad at searching, inserting, and deleting (except at the end).
- **Linear arrays**, or one dimensional arrays, are the most basic.
  - Are static in size, meaning that they are declared with a fixed size.
- **Dynamic arrays** are like one dimensional arrays, but have reserved space for additional elements.
  - If a dynamic array is full, it copies its contents to a larger array.
- **Multi dimensional arrays** nested arrays that allow for multiple dimensions such as an array of arrays providing a 2 dimensional spacial representation via x, y coordinates.

### Time Complexity

- Indexing: Linear array: `O(1)` , Dynamic array: `O(1)`
- Search: Linear array: `O(n)` , Dynamic array: `O(n)`
- Optimized Search: Linear array: `O(log n)` , Dynamic array: `O(log n)`
- Insertion: Linear array: n/a, Dynamic array: `O(n)`

## Linked List

### Definition

- Stores data with **nodes** that point to other nodes.

- Nodes, at its most basic it has one datum and one reference (another node).
- A linked list *chains* nodes together by pointing one node's reference towards another node.

### What you need to know

- Designed to optimize insertion and deletion, slow at indexing and searching.
- **Doubly linked list** has nodes that also reference the previous node.
- **Circularly linked list** is simple linked list whose **tail**, the last node, references the **head**, the first node.
- **Stack**, commonly implemented with linked lists but can be made from arrays too.
  - Stacks are **last in, first out** (LIFO) data structures.
  - Made with a linked list by having the head be the only place for insertion and removal.
- **Queues**, too can be implemented with a linked list or an array.
  - Queues are a **first in, first out** (FIFO) data structure.
  - Made with a linked list that only removes from head and adds to tail.

### Time Complexity

- Indexing: Linked Lists: `O(n)`
- Search: Linked Lists: `O(n)`
- Optimized Search: Linked Lists: `O(n)`
- Append: Linked Lists: `O(1)`
- Prepend: Linked Lists: `O(1)`
- Insertion: Linked Lists: `O(n)`

## Hash Table or Hash Map

### Definition

- Stores data with key value pairs.
- **Hash functions** accept a key and return an output unique only to that specific key.
  - This is known as **hashing**, which is the concept that an input and an output have a one-to-one correspondence to map information.
  - Hash functions return a unique address in memory for that data.

### What you need to know

- Designed to optimize searching, insertion, and deletion.
- **Hash collisions** are when a hash function returns the same output for two distinct inputs.
  - All hash functions have this problem.

    - This is often accommodated for by having the hash tables be very large.
- Hashes are important for associative arrays and database indexing.

### Time Complexity

- Indexing: Hash Tables: `O(1)`
- Search: Hash Tables: `O(1)`
- Insertion: Hash Tables: `O(1)`

## Binary Tree

### Definition

- Is a tree like data structure where every node has at most two children.
    - There is one left and right child node.

For a full binary-tree reference see Here

### What you need to know

- Designed to optimize searching and sorting.
- A **degenerate tree** is an unbalanced tree, which if entirely one-sided, is essentially a linked list.
- They are comparably simple to implement than other data structures.
- Used to make **binary search trees.**
    - A binary tree that uses comparable keys to assign which direction a child is.
    - Left child has a key smaller than its parent node.
    - Right child has a key greater than its parent node.
    - There can be no duplicate node.
    - Because of the above it is more likely to be used as a data structure than a binary tree.

### Time Complexity

- Indexing: Binary Search Tree: `O(log n)`
- Search: Binary Search Tree: `O(log n)`
- Insertion: Binary Search Tree: `O(log n)`

# Algorithms

## Algorithm Basics

# Recursive Algorithms

### Definition

- An algorithm that calls itself in its definition.
  - **Recursive case** a conditional statement that is used to trigger the recursion.
  - **Base case** a conditional statement that is used to break the recursion.

### What you need to know

- **Stack level too deep** and **stack overflow**.
  - If you've seen either of these from a recursive algorithm, you messed up.
  - It means that your base case was never triggered because it was faulty or the problem was so massive you ran out of alloted memory.
  - Knowing whether or not you will reach a base case is integral to correctly using recursion.
  - Often used in Depth First Search

# Iterative Algorithms

### Definition

- An algorithm that is called repeatedly but for a finite number of times, each time being a single iteration.
  - Often used to move incrementally through a data set.

### What you need to know

- Generally you will see iteration as loops, for, while, and until statements.
- Think of iteration as moving one at a time through a set.
- Often used to move through an array.

### Recursion Vs. Iteration

- The differences between recursion and iteration can be confusing to distinguish since both can be used to implement the other. But know that,
  - Recursion is, usually, more expressive and easier to implement.
  - Iteration uses less memory.
- **Functional languages** tend to use recursion. (i.e. Haskell)
- **Imperative languages** tend to use iteration. (i.e. Ruby)
- Check out this Stack Overflow post for more info.

### Pseudo Code of Moving Through an Array

```
Recursion                          | Iteration
-----------------------------------|-----------------------------------
recursive method (array, n)        | iterative method (array)
  if array[n] is not nil           |   for n from 0 to size of array
    print array[n]                 |     print(array[n])
    recursive method(array, n+1)   |
  else                             |
    exit loop                      |
```

## Greedy Algorithms

### Definition

- An algorithm that, while executing, selects only the information that meets a certain criteria.
- The general five components, taken from Wikipedia:
  - A candidate set, from which a solution is created.
  - A selection function, which chooses the best candidate to be added to the solution.
  - A feasibility function, that is used to determine if a candidate can be used to contribute to a solution.
  - An objective function, which assigns a value to a solution, or a partial solution.
  - A solution function, which will indicate when we have discovered a complete solution.

### What you need to know

- Used to find the expedient, though non-optimal, solution for a given problem.
- Generally used on sets of data where only a small proportion of the information evaluated meets the desired result.
- Often a greedy algorithm can help reduce the Big O of an algorithm.

### Pseudo Code of a Greedy Algorithm to Find Largest Difference of any Two Numbers in an Array.

```
greedy algorithm (array)
  var largest difference = 0
  var new difference = find next difference (array[n], array[n+1])
  largest difference = new difference if new difference is > largest difference
  repeat above two steps until all differences have been found
  return largest difference
```

This algorithm never needed to compare all the differences to one another, saving it an entire iteration.

# Search Algorithms

## Breadth First Search

### Definition

- An algorithm that searches a tree (or graph) by searching levels of the tree first, starting at the root.
  - It finds every node on the same level, most often moving left to right.
  - While doing this it tracks the children nodes of the nodes on the current level.
  - When finished examining a level it moves to the left most node on the next level.
  - The bottom-right most node is evaluated last (the node that is deepest and is farthest right of it's level).

### What you need to know

- Optimal for searching a tree that is wider than it is deep.
- Uses a queue to store information about the tree while it traverses a tree.
  - Because it uses a queue it is more memory intensive than **depth first search**.
  - The queue uses more memory because it needs to stores pointers

### Time Complexity

- Search: Breadth First Search: O(V + E)
- E is number of edges
- V is number of vertices

## Depth First Search

### Definition

- An algorithm that searches a tree (or graph) by searching depth of the tree first, starting at the root.
  - It traverses left down a tree until it cannot go further.
  - Once it reaches the end of a branch it traverses back up trying the right child of nodes on that branch, and if possible left from the right children.
  - When finished examining a branch it moves to the node right of the root then tries to go left on all it's children until it reaches the bottom.
  - The right most node is evaluated last (the node that is right of all it's ancestors).

### What you need to know

- Optimal for searching a tree that is deeper than it is wide.
- Uses a stack to push nodes onto.
  - Because a stack is LIFO it does not need to keep track of the nodes pointers and is therefore less memory intensive than breadth first search.
  - Once it cannot go further left it begins evaluating the stack.

### Time Complexity

- Search: Depth First Search: O(|E| + |V|)
- E is number of edges
- V is number of vertices

### Breadth First Search Vs. Depth First Search

- The simple answer to this question is that it depends on the size and shape of the tree.
  - For wide, shallow trees use Breadth First Search
  - For deep, narrow trees use Depth First Search

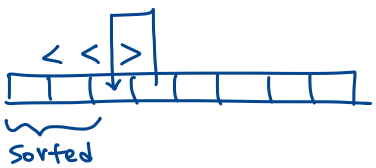
### Nuances

- Because BFS uses queues to store information about the nodes and its children, it could use more memory than is available on your computer. (But you probably won't have to worry about this.)
- If using a DFS on a tree that is very deep you might go unnecessarily deep in the search. See xkcd for more information.
- Breadth First Search tends to be a looping algorithm.
- Depth First Search tends to be a recursive algorithm.

# Sorting Algorithms

## Selection Sort

### Definition




- A comparison based sorting algorithm.
  - Starts with the cursor on the left, iterating left to right
  - Compares the left side to the right, looking for the smallest known item
    - If the left is smaller than the item to the right it continues iterating
    - If the left is bigger than the item to the right, the item on the right becomes the known smallest number

        - Once it has checked all items, it moves the known smallest to the cursor and advances the cursor to the right and starts over
    - As the algorithm processes the data set, it builds a fully sorted left side of the data until the entire data set is sorted
- Changes the array in place.

### What you need to know

- Inefficient for large data sets.
- Very simple to implement.

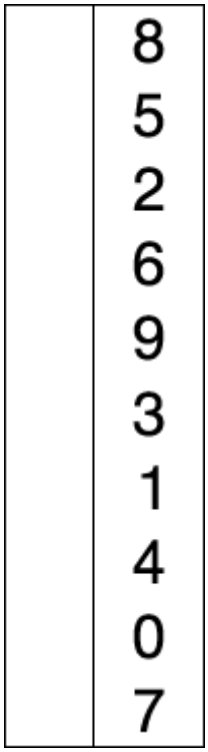
### Time Complexity

- Best Case Sort: `O(n^2)`
- Average Case Sort: `O(n^2)`
- Worst Case Sort: `O(n^2)`

### Space Complexity

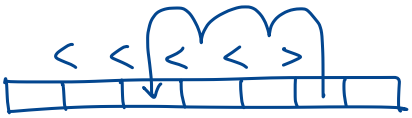
- Worst Case: `O(1)`

### Visualization




(source: Wikipedia, *Selection Sort*)

# Insertion Sort



### Definition

- A comparison based sorting algorithm.
  - Iterates left to right comparing the current cursor to the previous item.
  - If the cursor is smaller than the item on the left it swaps positions and the cursor compares itself again to the left hand side until it is put in its sorted position.
  - As the algorithm processes the data set, the left side becomes increasingly sorted until it is fully sorted.
- Changes the array in place.

### What you need to know

- Inefficient for large data sets, but can be faster for than other algorithms for small ones.
- Although it has an `O(n^2)` time complexity, in practice it is slightly less since its comparison scheme only requires checking place if it is smaller than its neighbor.

### Time Complexity

- Best Case: `O(n)`
- Average Case: `O(n^2)`
- Worst Case: `O(n^2)`

### Space Complexity

- Worst Case: `O(n)`

### Visualization

6 5 3 1 8 7 2 4

(source: Wikipedia, *Insertion Sort*)

# Merge Sort

### Definition

- A divide and conquer algorithm.
  - Recursively divides entire array by half into subsets until the subset is one, the base case.
  - Once the base case is reached results are returned and sorted ascending left to right.
  - Recursive calls are returned and the sorts double in size until the entire array is sorted.

### What you need to know

- This is one of the fundamental sorting algorithms.
- Know that it divides all the data into as small possible sets then compares them.

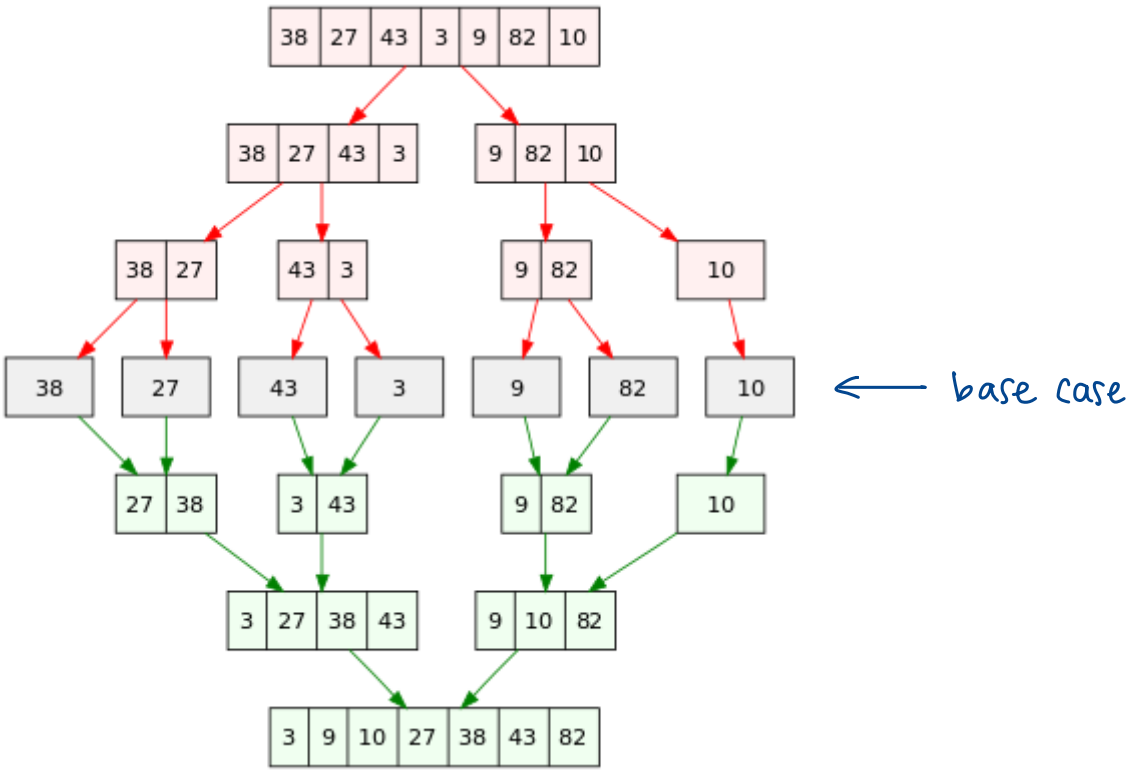
### Time Complexity

- Worst Case: `O(n log n)`
- Average Case: `O(n log n)`
- Best Case: `O(n)`

### Space Complexity

- Worst Case: `O(1)`

### Visualization




(source: Wikipedia, *Merge Sort*)

# Quicksort

### Definition

- A divide and conquer algorithm
  - Partitions entire data set in half by selecting a random pivot element and putting all smaller elements to the left of the element and larger ones to the right.
  - It repeats this process on the left side until it is comparing only two elements at which point the left side is sorted.
  - When the left side is finished sorting it performs the same operation on the right side.
- Computer architecture favors the quicksort process.
- Changes the array in place.

### What you need to know

- While it has the same Big O as (or worse in some cases) many other sorting algorithms it is often faster in practice than many other sorting algorithms, such as merge sort.

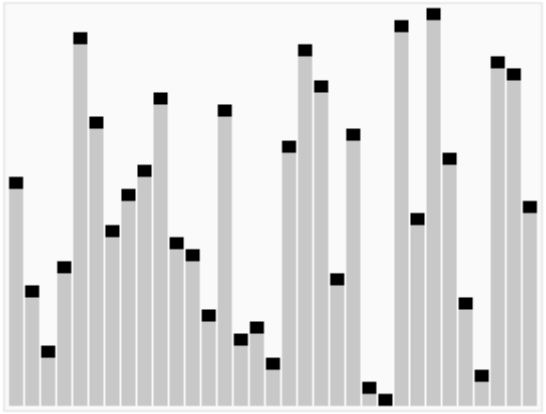
### Time Complexity

- Worst Case: `O(n^2)`
- Average Case: `O(n log n)`
- Best Case: `O(n log n)`

### Space Complexity

- Worst Case: `O(log n)`

### Visualization



(source: Wikipedia, *Quicksort*)

### Merge Sort Vs. Quicksort

- Quicksort is likely faster in practice, but merge sort is faster on paper.
- Merge Sort divides the set into the smallest possible groups immediately then reconstructs the incrementally as it sorts the groupings.
- Quicksort continually partitions the data set by a pivot, until the set is recursively sorted.

## Additional Resources

Khan Academy's Algorithm Course Graph Data Structure & Algorithms Data Structure Interview Questions Data Structure MCQ With Answers 10 Best Data Structures and Algorithms Books

~~heap~~

~~BST~~

~~BFS~~

~~DFS~~

~~KNN~~

~~k-means~~ & ~~farthest point sampling~~ (~~k-means++~~) & ~~weighted k-means~~

hash table

~~sorting~~

# Data Structures

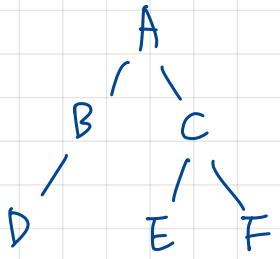
| | array | linked list | stack | queue | hash table | BST | heap |
|---|---|---|---|---|---|---|---|
| | sequential | node w/ pointer | LIFO<br>insert & remove head | FIFO<br>remove head & insert tail | key-value<br>hashing | 2 children/ node<br>left < node<br>node ≤ right | 'best' @ top<br>node < children |
| indexing | O(1) | O(n) | | | O(1) | O(log n) | *get min O(1) |
| search | O(n)<br>*O(log n) | O(n) | | | O(1) | O(log n) | *remove min O(log n) |
| insertion | O(1) | O(n) | | | O(1) | O(log n) | O(log n)<br>*heapify : O(n) |
| append/ prepend | | O(1)/ O(n) | | | | | |

# BFS

- find shortest path in graph
- queue (FIFO)
- slower than DFS & more memory-intensive
- O(V+E)

1. visit a vertex & mark as visited
2. enqueue children

```
graph = defaultdict(list)
visited = [False] * len(vertex)

queue = []
queue.append(s)

visited[s] = True

while queue:
    s = queue.pop()
    print(s)

    for i in graph[s]:
        if !visited[i]:
            queue.append(i)
            visited[i] = True
```



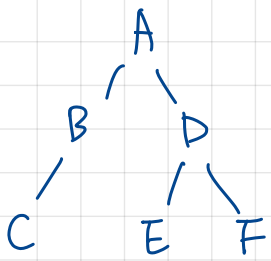

queue: ← A B C D E F ←

## DFS

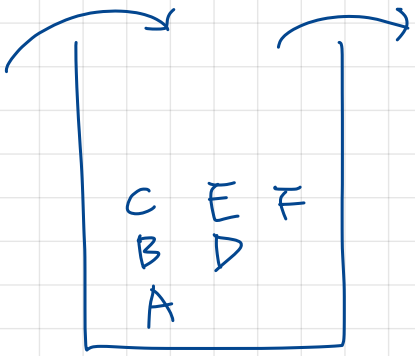
- stack (LIFO)
- O(|V| + |E|)

1. push visited vertices into stack
2. if X vertices, pop visited vertices

```
graph = defaultdict(list)
visited = set()

def DFS(v, visited):
    visited.add(v)
    print(v)

    for i in graph[v]:
        if i not in visited:
            DFS(i, visited)

DFS(s, visited)
```




stack :



# k-Nearest Neighbors

- distance metrics
  - Euclidean : $d(x,y) = \sqrt{\sum_i (x_i - y_i)^2}$
  - Manhattan : $d(x,y) = \sum_i |x_i - y_i|$
- advantages : easy to implement, few hyperparams
- disadvantages : computation-heavy & storage-heavy, X work well w/ high-dim, prone to overfitting
- $O(n \log n)$

```
dic = []
for i, example in enumerate(data):
    dist = distance_fn(example, query)
    dic.append((dist, i))

sorted_dic = sorted(dic)

knn = sorted_dic[:k]
knn_labels = [data[i][-1] for d,i in knn]

return choice_fn(knn_labels)
```

- $O(n \log k)$ : min heap of size k

```
heap = []

for point in points:
    dist = distance_fn(query.point)
    if len(heap) == k:
        heappq.heappushpop(heap, (dist, point))
    else:
        heappq.heappush(heap, (dist, point))

return [point for (dist, point) in heap]
```

- O(n) : heapify

```
heap = []

for point in points:
    dist = distance_fn(query, point)
    heap.append((dist, point))

heappq.heapify(heap)

knn = []
for i in range(k):
    dist, point = heappq.heappop(heap)
    knn.append(point)

return knn
```

# K-means Clustering

- unsupervised ML : group unlabeled dataset into dif clusters

1. randomly initialize K points (cluster centroids = means)
2. categorize each item to its closest centroid
3. update centroid to avg of all the items in its cluster
4. repeat 2-3 for a given # iterations

```
def init_centroids(data):
    random_idx = np.random.permutation(data.shape[0])
    centroids = data[random_idx[:K]]
    return centroids

def compute_centroids(data, labels):
    centroids = np.zeros((K, data.shape[1]))
    for i in range(K):
        centroids[i,:] = np.mean(data[labels==i,:], axis=0)
    return centroids

def compute_dist(data, centroids):
    dist = np.zeros((data.shape[0], K))
    for i in range(K):
        dist[:,i] = np.square(np.linalg.norm(data-centroids[i,:], axis=1))
    return dist

def find_closest_cluster(dist):
    return np.argmin(dist, axis=1)
```

```
def fit(data):
    centroids = init_centroids(data)
    for i in range(max_iter):
        distances = compute_dist(data, centroids)
        labels = find_closest_cluster(distances)
        centroids = compute_centroids(data, labels)

    return centroids

def predict(X, centroids):
    dist = compute_dist(X, centroids)
    return find_closest_cluster(dist)
```

- farthest point sampling

: choose centroids that are far away from one another

→ ↑ chance of initially picking up centroids that lie in dif clusters

1. randomly select 1 centroid
2. Compute (min) distance of all points & chosen centroid(s)
3. select point w/ max distance as the next centroid
4. repeat 2-3 until k centroids are sampled

- K-means++

: above more likely to use outliers as initial centroids

→ choose next center by $P(\text{dist}^2(x, \text{prev centroid}))$

$$= \frac{D^2(x, pc)}{\sum_i D^2(x_i, pc)}$$

# Weighted K-means Clustering

- when all datapoints X have equal importance
- compute_centroid() : use weighted mean

$$\text{mean} = \frac{1}{\sum_i w_i} \sum_i w_i x_i$$

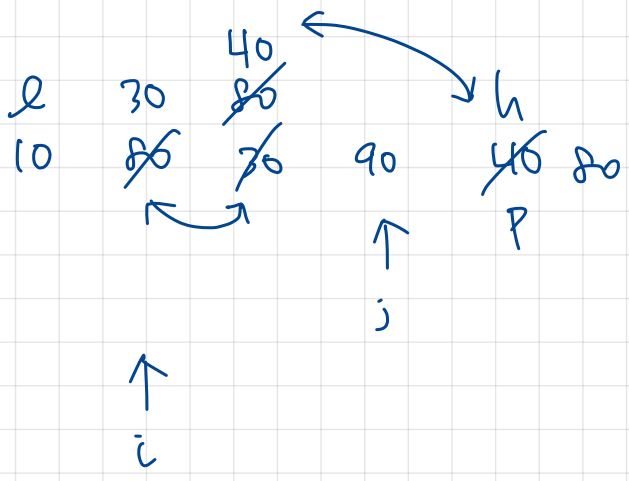
# Sorting

| | best | avg | worst | memory | method |
|---|---|---|---|---|---|
| quick | $n\log n$ | $n\log n$ | $n^2$ | $\log n$ | partitioning w/ pivot |
| merge | $n\log n$ | $n\log n$ | $n\log n$ | $n$ | divide-sort-merge |
| heap | $n\log n$ | $n\log n$ | $n\log n$ | 1 | heapify & extract max |
| insertion | $n$ | $n^2$ | $n^2$ | 1 | place it in the right place |
| tim | $n$ | $n\log n$ | $n\log n$ | $n$ | insertion & merging |
| selection | $n^2$ | $n^2$ | $n^2$ | 1 | iteratively find next smallest |
| bubble | $n$ | $n^2$ | $n^2$ | 1 | swap like crazy |
| count | $n+k$ | $n+k$ | $n+k$ | $k$ | Cumulative count for reserving space for each item *need to know the range |

# Quicksort

1. Choose a pivot (= last element in partition)
2. put all elements smaller than pivot to the left of pivot
   & all " greater " right "
3. put pivot in the correct position

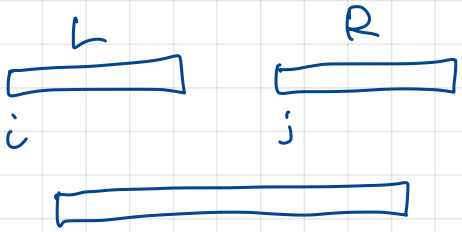
```
def partition(array, l, h):
    pivot = array[h]          rightmost

    i = l-1                   pointer for greater element

    for j in range(l, h):
        if array[j] <= pivot:
            i += 1
            array[j], array[i] = array[i], array[j]      swap w/ greater element

    array[i+1], array[h] = array[h], array[i+1]      swap pivot w/ greater element

    return i+1                position where partition is done

def quicksort(array, l, h):
    if l < h:
        pivot = partition(array, l, h)
        quicksort(array, l, pivot-1)      left of pivot
        quicksort(array, pivot+1, h)      right of pivot
```



# MergeSort

1. divide an array into 2 halves
2. sort each half
3. merge sorted halves back tgt

```
def mergesort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L, R = arr[:mid], arr[mid:]
        mergesort(L)
        mergesort(R)

        i,j,k = 0,0,0
        while i < len(L) and j < len(R):
            if L[i] <= R[j]:
                arr[k] = L[i]
                i+=1
            else:
                arr[k] = R[j]
                j+=1
            k+=1

        while i<len(L):
            arr[k] = L[i]
            i+=1
            k+=1
        while j< len(R):
            arr[k] = R[j]
            j+=1
            k+=1
```




check if any element was left

# HeapSort

1. heapify array into max heap
2. repeat until heap has 1 element:
   ① swap root (max) w/ last element
   ② remove last element of heap
   ③ heapify remaining → find the next largest
3. reverse the array

```
def heapify(arr, N, i):
    largest = i
    l = 2i+1
    r = 2i+2

    if l<N and arr[largest] < arr[l]:
        largest=l

    if r<N and arr[largest] < arr[r]:
        largest=r

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, N, largest)

def heapsort(arr):
    N=len(arr)

    for i in range(N//2-1, -1, -1):
        heapify(arr,N,i)

    for i in range(N-1,0,-1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr,i,0)
```

ensure node > children

# InsertionSort

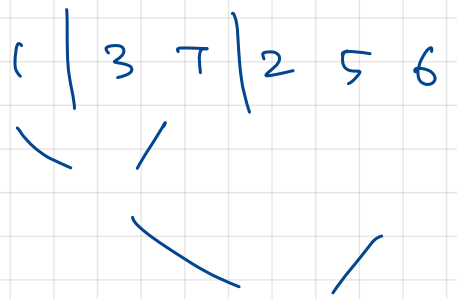
1. iterate over the array:
   ① if current < predecessor,
      compare it to elements before
   ② move greater elements 1 position up
      to make space for swapped element

- left of current : sorted
- right " : X sorted
- values from unsorted picked & placed @ correct position in sorted part

```
for i in range(1, len(arr)):
    key = arr[i]

    j=j-1
    while j>=0 and arr[j]> key :
        arr[j+1] = arr[j]
        j-=1
    arr[j+1] = key
```

# TimSort

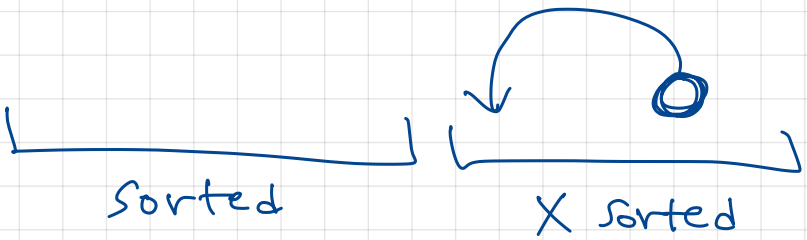
- hybrid sorting algorithm from merge sort & insertion sort
- default for Python's sorted(), list.sort()
- exploit existing order in data → minimize #comparisons & swaps
  1. divide array into smaller subarrays (sorted)
  2. merge using modified merge sort algorithm



# Selection Sort

1. repeat until entire list is sorted
   ① select smallest element from unsorted part
   ② swap w/ 1st element of unsorted part

```
for i in range(len(arr)):
    min_idx = i
    for j in range(i+1, len(arr)):
        if arr[min_idx] > arr[j]:
            min_idx = j
    arr[i], arr[min_idx] = arr[min_idx], arr[i]
```



# BubbleSort

1. repeat until data is sorted :

   ① compare adjacent elements
   & place higher one @ right

```
for i in range(len(arr)):
    swapped = False

    for j in range(0, len(arr)-i-1):    last i elements already sorted
        if arr[j] > arr[j+1]:
            arr[j], arr[j+1] = arr[j+1], arr[j]
            swapped = True

    if !swapped:
        break
```