Searching Algorithm

Binary Search

Description: Divide and conquer on a **monotonic** increasing array; compare the median with the item, if the median > item, search in the right half, else if median < item, search in the left half, else median == item and return median

Invariant: A[begin] <= key <= A[end], i.e the key is in the range of the array.

At k^{th} iteration, end – begin = $n/2^k$

Runtime Complexity: O(logn)

Preconditions:

- Array is of size N

- Array is sorted

Pros: Fast searching in **sorted** arrays

Cons: Cannot be used on unsorted arrays

Peak Finding

Description: Divide and conquer on unsorted array; compare the neighbour of the median, if peak return median, else if right > median, search in right half, else if left > median, search in left half

Invariant: If we recurse in the right half, then there exists a peak in the right half.

Runtime Complexity: O(logn)

2D n x m array Runtime Complexity:

Simple – O(nlogm)

Careful – O(n + m)

Pros: Fast in finding local maximums

Cons: Does not find global maximum, finding global maximum may require O(n) runtime.

Quick Select

Description: Divide and conquer on an unsorted array; compare the index of the pivot, when searching for k^{th} smallest item, if k < index of pivot, search for k^{th} element in left half, else if k > index of pivot, search for $(k – pIndex)^{th}$ element in right half; else if k == index of pivot, return index

Invariant: At the end of j iterations,
for all i >= high, A[i] > pivot,
for all i < high, A[i] < pivot

Runtime Complexity:

Best/Average case - O(n)

Worst case – O(n^2)

(if key chosen is always min or max)

Sorting Algorithm

Make sorting algorithm stable

- **associate original indices of each key with the key**
- simple way is to create auxillary array of indices which swaps will be performed on too
- use this indices array to disambiguate elements with equal keys
- will **make sorting algo not In-Place anymore**

Bubble Sort

Description: Repeatedly compares and swaps two adjacent elements if they are in wrong order

Invariant: At the end of j iterations, the biggest j items are correctly sorted in the final j positions of the array

Runtime Complexity:

Best case - O(n) (Already sorted)

Average case - O(n^2)

(Assume input are chosen at random)

Worst case - O(n^2) (Inversely sorted)

Space Complexity: 0(1)

Stability: Stable

In place: Yes

Selection Sort

Description: Repeatedly iterates through the array to find the smallest element, then swap it with the first element

Invariant: At the end of j iterations, the smallest j items are correctly sorted in the first j positions of the array

Runtime Complexity:

Best case - O(n^2)

Average case – O(n^2)

Worst case - O(n^2)

(**Never makes more than O(n) swaps**)

Space Complexity: 0(1)

Stability: Unstable

In place: Yes

Insertion Sort

Description: Iterates through the array and sorts the element by inserting it into the sorted elements at the front of the array

Invariant: After j iterations, the first j items in the array are in sorted order

Runtime Complexity:

Best case - O(n) (Already sorted)

Average case - O(n^2)

(random permutation)

Worst case - O(n^2) (Inversely sorted)

Space Complexity: 0(1)

Stability: Stable

In place: Yes

Merge Sort

Description: Repeatedly split array into two halves, sort the two halves and merge the two sorted halves

Invariant: At each recursive call, the length of the subarray is strictly decreasing till size <= 1, in which the algorithm will terminate.

Runtime Complexity:

Best case - O(n log n)

Average case – O(n log n)

Worst case - O(n log n)

Space Complexity:

O(nlogn) (**Top-down approach**) **OR**

O(n) (**Bottom-up approach**)

Stability: Stable

In place: No

Quick Sort (3 way)

Description: Selects a pivot at random and compares each element to the pivot (smaller on left bucket, larger on right bucket, equal in the middle) Recurse on left and right bucket till array is sorted

Invariant: At end of every iteration,
for all i >= high, A[i] > pivot,
for all 1 < j < low, A[j] < pivot

Runtime Complexity:

Best case - O(n log n) (Already sorted)

Average case - O(n log n)

(Random pivot/median)

Worst case - O(n^2)

(If key is consistently min or max)

Space Complexity: O(log n)

Stability:

Not stable (if using in-place partition)

Stable (if using extra space)

In place: Yes

Note: Using more pivots for partitioning does not actually improve the Asymptotic Runtime of quicksort (Still O(nlogn)). Although in practice 2 and 3 pivots quicksort does run faster (in real time).

Search Trees

Binary-trees

Description: A tree with at most two children, namely left and right child

Invariant: The children in the left-subtree will always be smaller than the parent, the children in the right-subtree will always be larger than the parent

Operations:

- **height**, **searchMin**, **searchMax**, **search**, **insert** {all O(h)}
- **in-order** (left, this, right) (O(n)}
- **pre-order** (this, left, right) (O(n)}
- **post-order** (left, right, this) (O(n)}
- **level order** (O(h)}
- **successor / predecessor** {O(h)}

(search for key, result > key
? result
: continue searching for successor of result)

- **delete** {O(h)}

Pros: Fast inserts, search and delete operations.

Cons: Can be unbalanced and query operations may take up to O(n)

AVL-trees

Description: Like binary trees but height balanced (at most height of 2logn, at least $2^{h/2}$ nodes and at most 2^{h+1} - 1 nodes).

Invariant: Difference between height of left sub-tree & right sub-tree <= 1

Operations:

(all operations run in O(logn))

Left and right rotation {O(1)}

Delete:

- may cause O(logn) rotations

After insertion if tree is left heavy:

- Left subtree is **balanced**: Right-rotate
- Left subtree is **left heavy**: Right-rotate
- Left subtree is **right heavy**: Left-rotate then right rotate
- Opposite for right heavy situation

(**Worst case**: 2 rotations)

Pros: Dynamic, self-balancing, relatively fast query methods {O(nlogn)}

Cons: Limited methods

Tries

Description: String-based data structure storing characters of a string such that the root to leaf path represents strings

Invariant: Each node has 256 possible children due to 256 unique ascii values

Operations:

- **search** (Prefix or pattern) {O(L)}
- **insert** {O(L)}

Note - Runtime does not depend on size of total text

Pros: Suitable data structure for storing strings

Cons: Tend to use more space as it has more nodes

kd-Trees

Description: 2-d tree, where each node represents a rectangle of a plane. A node has 2 children which divides the rectangle into 2 pieces either vertically or horizontally.

Operations:

- Build: At every iteration, find a random node then use it to partition and recurse into the 2 partitions {O(nlogn)}
- Find min/max of 1 coordinate {O(sqrt(n))}

(a,b) / B-trees

Description:

In an (a, b)-tree, a and b are parameters where 2 <= a <= (b+1)/2.
a and b refers to the min and max # of children an internal node can have.
It is a self-balancing tree that allows **nodes to have more than one child** and **nodes to have more than one key**, with a minimum degree "t" denoting the number of keys in a node

Invariant: All the **leaves are at the same level**, each node has **at least t-1 keys and at most 2t-1 keys sorted in increasing order**, and the **number of children of a node is the number of keys + 1**

Operations:

Most if not all has O(h) runtime complexity

- **search**

(start from root, then recursively traverse down)

- traverse

(like in order in binary tree)

- **insertion**

(split if node becomes too full)

- **deletion**

(merge if nodes not enough children, then split again if nodes have too much children after merging)

Pros: Self-balancing and dynamic. Well suited for storage systems that read and write relatively large blocks of data, thus good management on space.

Searching is very fast as O($\log_b n$)

Cons: Not as efficient as other balanced trees as inserting and deletion could involve moving many keys

Augmented trees

Steps to Augment Data Structure:

1- Choose underlying data structure

2- determine additional info needed

3- maintained info as data structure is modified

4- develop new operations using new info)

Order statistics trees (Augmented AVL tree)

Augment: Store weight (number of children) of subtree in each node

Purpose: Finds rank and k^{th} smallest key in O(logn) time

Real life examples:

- find the 10th tallest student in the class.
- determine the percentile of Johnny's height.

Is Johnny in the 10th percentile or the 90th percentile?

Intervals trees (Augmented AVL tree)

Augment: Store interval in each node to and the max endpoint of subtrees.

Purpose: Find an interval that contains k **(O(logn))** and Finds ALL intervals that contains k.

(O(jlogn) where j is the number of overlapping intervals that contains k)

Real life example:

- Find a cell tower that covers a certain location

Tournament Tree (Augmented BST)

Augment: Start with perfectly balanced BST with each contender at leaf nodes.

Purpose: Provides a quick way(least comparisons needed) to determine a winner among n contenders. **{O(nlogn)}**

Real life examples: Tennis matches, finding best chicken rice among n plates of chicken rice

Range trees (Augmented Binary Search Tree)

1-dimensional:
Augment: Store all the **points at leaves** (internal nodes store only copies) and each internal node stores the **max of any leaf in the left subtree**
{Building the tree: O(nlogn), Space: O(n)}
Purpose: Efficient range queries (important for databases) in O(k + logn) time (where k is the number of items found and log n is time to find split node)
Real life example:
- Find me everyone between ages 22 and 27

2-dimensional:
Augment: Build a x-tree using only x coordinates. For every node in the x-tree, build a y-tree out of nodes in subtree using only y-coordinates
{Building tree: O(nlogn), Space: O(nlogn),
Rotation : O(n)}
Purpose: Efficient range query in 2d space.
{O($\log^2$n + k)}
Real life example:
- Find me all the good restaurant in my area.

Hashing and HashTables

Description: Uses hash functions to map each value to a bucket with a unique key in a symbol table
Generally assume hash function uses O(1) to compute.
Simple Uniform Hashing Assumption (SUHA)
Every key is equally likely to map to every bucket and keys are mapped independently.
Key must be **IMMUTABLE** (e.g Integer, String, etc.)
A good hash function enumerates all possible buckets, creating permutations of {1..m} and has SUHA
No successor/predecessor queries!
Must override .equals and hashCode methods

Good Hash Function
1. h(key, i) enumerates all possible buckets
    - hash function is permutation of {1…m}
    - every bucket there is some value i that will map to it
2. Uniform Hashing Assumption
    - Every key is equally likely to be mapped to every permutation, independent of other key

Chaining
Description: Total **m buckets**, each containing a **linked list of n items**, thus **total space is O(n+ m)** All items mapped to the same slot are stored in a link list.
load(average # items/bucket) = n/m
Operations:
- **Insert(Key k, Value v)**
Calculate h(key), then add (key, value) to linked list where h() is the hash function.
Worse case **O(1+ cost(h))** (assuming allow duplicates)
- **Search (Key k)**
Calculate h(key), then search for (key, value) in linked list
Worse case **O(n + cost(h))**
- **Delete(Key k)**
- **Contains(Key k)**
- **Size()**
Average case:
Expected search time = 1 + n/m {O(1)}
Expected max cost of inserting n items = O(logn)
(m = number of buckets, n = number of elements)
Pros: Searches efficiently (by not using comparison-based searching), able to add new items to hash table even if m = = n
Cons: Space might be big **{O(n+m)}**, searching is relatively less efficient if hashing function is not good.

Open addressing
Description: On collision, probe a sequence of buckets until an empty one is found. The probe sequence specifies order in which cells are examined.
Hash functions becomes h(key, i), where i is the number of collisions
Probing Methods:
- Linear probing (**does not fulfil Uniform Hashing Assumption**)
Sequence of bucket is in order
(Linear probing is **slow due to clusters {O(logn)}**
(**Fast practically due to caching:** cheap to access nearby cells)
Operations:
- insert
- search
- delete
Set bucket to deleted (tombstone value), **overriding when inserting** and **ignored when searching**
Expected cost of operations = 1 / (1- α), where α = n / m

Pros: Saves space, rarely allocate memory, better cache performance
Cons: More sensitive to the choice of hash function and load.

Probability Theory

$E[X] = e_1p_1 + e_2p_2 + \dots + e_np_n$
E[A+B] = E[A]+E[B]

Coin Flips
In two coin flips: I expect one heads when flipping an unfair coin:
Pr(heads) = p
Pr(heads) = 1 – p
E[X] = (p)(1) + (1 – p) (1 + E[X]) => E[X] = 1/p

Simple recurrences

(Hint: to solve recurrences, can guess and verify, draw recursion tree or use Master Theorem, solve by induction)

T(n) = T(n/2) + 1 {O(logn)}
T(n) = 2T(n/2) + 1 {(O(n)}
T(n) = T(n/2) + n {O(n)}
T(n) = √nT(√n)+ √n {O(n)} (Sieve of Eratosthenes)
T(n) = 2T(n/2) + n {O(nlogn)}
T(n) = 2T(n/2) + O(n log n) {O(nlog^{2}n)}
T(n) = 1 + T(n-1) + T(n-2) / T(n-1) + T(n-2)
{O(Φ^n) (tight bound) / O(2^n) (loose bound)}

Sequences/Math Stuff

1+2+3+…+n = n(n+1)/2 = O(n^2)
logN! = O(NlogN) (Sterling's approximation)
n+n/2+n/4+…+n/$2^{\log n}$ = 2n
1+1/2+1/3+1/4+…1/n = O(logn) (Harmonic series)
1+2+4+…+n = O(2n-1)
1+2^2+3^2+…+n^2 = (n/6)(n+1)(2n+1) = O(n^3)
log(ab) = loga + logb
log(a/b) = loga - logb
h(n) = f(n)$^{g(n)}$ in this case **note the constant in g(n)!**
e.g h(n) = $n^{\log n} \neq n^{2\log n}$

	Arithmetic Progression (AP)	Geometric Progression (GP)
Formula for term u_n	$u_n = a+(n-1)d$	$u_n = ar^{n-1}$
Formula for sum S_n	$S_n = \frac{n}{2}(2a+(n-1)d)$ $= \frac{n}{2}(a+l)$	$S_n = \frac{a(1-r^n)}{1-r}$ or $\frac{a(r^n-1)}{r-1}$

Important definitions

Invariant: The relationship between variables that is always true
Loop Invariant: The relationship between variables that is true at the beginning (or end) of each iteration of the loop
Amortized Complexity: A way to express time complexity instead of average time complexity, to cater for algorithms that have a very bad time complexity once in awhile

Miscallaneous

Permutation Generation Algorithm
Sorting Shuffle
Idea: Assign each element in the array a random "key"from 0 to 1 then sort using this key
Runtime: O(nlogn) (due to sorting)
Cons: As array length increase, we expect duplicate numbers to occur more frequently. Solution might not scale well for large n as randomness of permutations degrade

NaiveShuffle
Idea: Have 2 arrays, for i = 0 to n – 1, generate a random number **R** from 0 to n-1, assign tempArray[i] = originalArray[R], throw picked ones to the back of the original array.
Runtime: O(n)

Sorting Jumble
Do in this order:
1. Selection sort
- Smallest j items will be sorted in to the smallest j slots after j iterations
- Index of smallest j element will swap with the unsorted j element
2. Insertion Sort
- Compared with **last few elements** of unsorted array, they will be **untouched**
- Find the smallest index of these untouched elements, **anything above this element should be sorted**
3. Bubble Sort
- Largest j elements will be in the last j slots in the sorted array
4. QuickSort
- Find pivot element, anything less than pivot should appear before it in the array, anything larger than pivot should appear after pivot
5. MergeSort
- Keep splitting into halves, check first half to see if its sorted, if not check second half.

Note on recursion
It is always better to recurse into the "smaller" recursion first!!
  - Less to store on call stack
  - Minimizes depth of call stack to deal with smaller cases first.

Sorting Optimization
For **merge** sort:
  - Consider switching to insertion sort when Array size is < 1024
  - Relies on the fact that insertion sort works fast on smaller arrays
For **quick** sort:
  - Consider switching to insertion sort when array size is small i.e we halt recursion (down to size of 1) early
  - Relies on the fact that insertion sort works very fast on almost sorted array.

BST Rebuilding
- Can rebuild if we are given:
  - Inorder + Preorder/Postorder/Level-order
  - Preorder

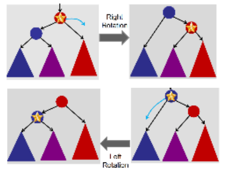
Psuedocodes
```
partition(A[], n, pIndex) {
  pivot = A[pIndex]
  swap(A[0], A[pIndex])
  low = 1
  high = n
  while (low < high) {
      while(A[low] < pivot) && (low < high) do low++
      while(A[high] > pivot) && (low < high) do high––
      if(low < high) then swap(A[low], A[high])
  }
  swap(A[0], A[low–1])
  return low–1
}

binarySearch(A, key, n) {
  begin = 0
  end = n–1
  while begin < end {
    mid = begin + (end – begin) / 2
    if key <= A[mid] then
      end = mid
    else
      begin = mid + 1
  }
  return(A[begin] == key) ? begin : –1
}

quickSelect(A[], n, k) {
  if(n == 1) then return A[0]
  else Choose random pivot index pIndex {
    p = partition(A, n pIndex)
    if(k==p) then return A[p]
    else if(k < p) then
      return quickSelect(A[1..p–1], k)
    else if(k > p) then
      return quickSelect(A[p+1], k – p)
}

successor(Node){
  if (Node.right != null)
    return Node.right.searchMin();
  parent = Node.parent;
  child = this;
  while ((parent != null) && (child == parent.right)) {
    child = parent;
    parent = child.parent;
  }
  return parent;
}

right–rotate(v) { // assume v has left != null
  w = v.left
  w.parent = v.parent
  v.parent = w
  v.left = w.right
  w.right = v
}
```



Input order → / Algorithm ↓	Random	Sorted: Ascending	Sorted: Descending	Nearly Sorted: Ascending	Nearly Sorted: Descending	Homogeneous (i.e. identical elements)
(Opt) Bubble sort	O(N²)	O(N) - best	O(n²)	O(n²)	O(n²)	O(n)
(Min) Selection Sort (least number of swaps)	O(n²)	O(n²)	O(n²)	O(n²)	O(N²)	O(n²)
Insertion Sort	O(n²)	O(n) - best	O(N²)	O(n)	O(n²)	O(n)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(N log N)	O(nlogn)	O(nlogn)
(Naive) Quick Sort*	O(nlogn)	O(N²)	O(n²)	O(n²)	O(n²)	O(n²)
(Rand) Quick Sort	O(N log N)	O(nlogn)	O(nlogn)	O(nlogn)	O(nlogn)	O(n²)

* Non-random Quick Sort which always chooses first element as pivot

Hashing and HashTables (Continued)

Table resizing

Description: Start with a small constant table size, then grow and shrink the table as necessary.
To **resize**: Cost $O(m_1 + m_2 + n)$
- Choose a new table size m
- Choose a new hash function (since hash function depends on table size)
- For each item in old has table, compute new hash function and copy item to new bucket

Ideally:
- if (n = = m), then grow, m = 2m
- if (n < m/4), then m = m/2
- Every doubling -> at least m/2 new items were added.
- Every shrink -> at least m/4 items were deleted.

Operations:
Insert amortized cost : O(1)
Amortized analysis of inserting k elements:
- Deferred dollars: O(k) (to pay for resizing)
- Immediate dollars: O(k) for inserting elements in table
- Total(Deferred + Immediate): O(k)

Pros: Space efficiency, relatively low amortized cost
Cons: Larger the size, the greater the cost of resizing

Fingerprint HashTable (FHT)

Description: Similar to hash tables but instead of storing keys, 0/1 bits information is stored
Pros: Reduced space: only 1 bit per slot. **No false negatives.**
Cons: Collision might result in **false positives**, thus increased space to reduce chance of collision.
Probability of false positive: $1 – (1/e)^{n/m}$
Minimum number of buckets: n / ln(1/1-p)
E.g We want false positive < 5%: m >= n/ln(1/1-0.05)

Bloom Filter (Extension of FHT)

Description: Similar to fingerprints but uses **k** hash functions to map item to k buckets of the table Checks all k buckets in order to determine the status of item
Pros: Compared to fingerprints, less chance of collision, thus saving table space
Cons: When k gets too large, space complexity and probability of false positive might become worse
(Assuming each table slot is independent) ->
Probability of false positive: ($1 – (1/e)^{kn/m})^k$
Minimum number of buckets: 2n / ln(1/1-√p)
Optimal value of k = m/nln2
Intersection: Bitwise **AND** of 2 bloom filters {O(m)}
Union: Bitwise **OR** of 2 bloom filters {O(m)}
Deletion:
1) Store count of items which hash to each fingerprint.
2) Use another FHT/Bloom Filter to keep track of deleted entries.
Deletion Operation will cause **FALSE NEGATIVES** to occur for both FHT and Bloom Filter.

Merkle Tree

Description: Used for efficient data verification. Stores hashes instead of full files. Much better space usage and very fast searching.
Idea: When a bunch of files needs to be compared to another bunch of files, we can just hash both of these bunch of files into one single hash code and compare them instead of hashing and comparing them one by one.
The root will be the hash of all the files and then it branches out into smaller and smaller subsections with leafs as individual files with their own hashcodes.

Graph

Terminology

Connected: Every pair of nodes is connected by a path.
Disconnected: Some pair of nodes is not connected by a path.
Degree of node: Number of adjacent edges of a node.
Degree of graph: Maximum degree of all nodes in graph.
Diameter: Maxumum distance between two nodes, **following the shortest path.**

Special Graphs

Star – One central node, all other nodes connect to central node
(Diameter: 2, Degree: n-1)
Clique (Complete graph) – All pairs connected by edges
(Diameter: 1, Degree: n-1)
Line (or path) – (Diameter: n-1, Degree: 2)
Cycle – (Diameter: n/2 or n/2 -1, degree 2)
Bipartite – Nodes divided into two sets with no edges between nodes in the same set
(Diameter: Max n-1)
* Fastest way to determine whether a graph is bipartite is by using graph colouring.
* Even cylces are bipartite, odd cylces are not bipartite.

Representing Graphs

Basic rule: If graph is dense (many edges) then use an adjacency matrix; else use an adjacency list
Dense: |E| ≈ O(V^2)
(Maximum number of edges in an undirected graph = n(n-1)/2 {O(V^2)}) (Happens in Cliques)

Adjacency List

Nodes: stored in an array
Edges: linked list per node
Memory Usage:
array size: |V|
linked list size: |E|
Total: O(V + E)
({O(V) for cycles}, {O(V^2) for cliques})
Use cases**:**
Fast query
- find me any neighbor {O(1) – poll first element}
- enumerate all neighbors {O(n) – iterate number of neighbors}

Slow query
- are v and w neighbors? {O(n) – iterate through whole list}

Adjacency Matrix

Nodes: stored in a n x n 2D matrix, in which n is the number of nodes
Edges: pair of nodes (for instance, A[a][b] represents edge between nodes a and b in graph A)
Property:
A^n = length n path (Not the most efficient way)
Memory Usage:
array size: |V| * |V|
Total: ALWAYS O(V^2)
Use cases:
Fast query
- are v and w neighbors? {O(1) – lookup}

Slow query
- find me any neighbor of v {O(V) – iterate through v columns}
- enumerate all neighbors {O(V) – iterate through v columns}

Reverse Adjacency List

If given a graph where we know that the graph is almost fully connected and is just missing a k edges, consider storing the missing edges as an adjacency list instead. This is more space efficient than storing the entire Adjacency list since it will only take up O(k) space compared to O(V^2) space which is what adjacency list and adjacent matrix will give.

Searching a graph (BFS & DFS)

- Visits every node in graph
- Visits every edge in graph
- **DOES NOT** visit every path in graph
- Both BFS and DFS performance **degrades to O(V^2)** when using adjacency matrix instead of adjacency list.

Breadth-First Search (BFS)

Results in a **shortest path** tree. Finds minimum number of hops to a node from a given node, DOES NOT find min distance.
Description: Explores level by level without back tracking
Runtime Complexity: O(V + E)
Start from a vertex v, every V is added to frontier once {O(V)}
Each v.nbrlist is enumerated once when v is removed from frontier {O(E)}
Data structure: Queue

Depth-First Search (DFS)

Results in a parent graph tree (**NOT** shortest path)
Description: Follow path till a dead end, backtrack till a new edge is found and recursively explore without repeating vertices
Runtime Complexity: O(V + E)
Calls DFS-visit only once per node {O(V)}, and in the DFS-visit, each neighbor is enumerated {O(E)}
Data structure: Stack

Single Source Shortest Path (SSSP) Algorithms

Solves:
- What is the shortest path from S to D.
- What is the shortest path from S to every node.

CAUTION: DO NOT USE BFS TO FIND SHORTEST PATH!
Overarching Idea: Triangle Inequality
Shortest path from S to C will always be less than or equal to the shortest path from S to A and from A to C.
δ(S, C) <= δ(S, A) + δ(A, C)

Bellman-ford

Description: A simple algorithm that relaxes all the edges v - 1 times. At each iteration, check for outgoing edges of each node and relax each edge. Terminates when an entire sequence of |E| relax operations have no effect.
Invariant:
After i iterations, the i hop estimate in the shortest path will be correct.
Runtime Complexity: O(EV), $O(V^2 + VE)$ if adjacency matrix is used.
Notes:
- Detecting negative weights:
Run Bellman-Ford for |V| times
If estimate changes in last iteration, then there exist negative weight cycle.
- If constant weight graph, use BFS

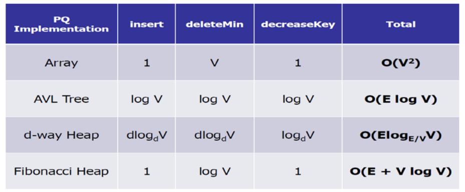
Trees

Description: **Only one possible path** from one node to another node, assuming:
i) weighted edges
ii) Positive or negative weights
iii) Undirected tree
Undirected tree: A graph with no cycles
Rooted tree: A tree with a special designated root note
Recursive definition of tree: A node with zero, one or more sub-trees
Algorithm: Relax edges in DFS or BFS pre-order
Runtime Complexity: O(V)
(In this case, O(V) = O(E) since E = V - 1 in trees)

Dijkstra's Algorithm

Works for **non-negative weights** graphs. Technically is a BFS using PQ as DS.
Description: Edges are relaxed in the "right order" and each edge is only relaxed once.
Maintain distance estimate for every node
Algorithm:
Repeat:
- Consider vertex with minimum estimate.
- Add vertex to shortest-path-tree. (Mark it as done)
- Relax all outgoing edges.

(Stop when destination is dequeued from PQ)
Invariant: Every "finished" vertex has correct estimate.
Intuition - Extending a path does not make it shorter! (Since edges are >= 0)
Data Structure: Priority queue
AVL tree to store edges to relax, and a **HashTable** to map keys to location in tree
Runtime Complexity:
insert/deleteMin: lVl times each (Each node added to PQ once)
relax/decreaseKey: lEl times (Each edge relaxed once)
PriorityQueue ops: O(logV) (For AVL Tree implementation)
Total: **O((V + E)logV) = O(ElogV)**
Note: Reweighting edges weight by adding a constant factor may not return shortest path since relative weight are changed, but multiplying a constant factor have no effects on relative weight -> Returns shortest path.



PQ Implementation	insert	deleteMin	decreaseKey	Total
Array	1	V	1	**$O(V^2)$**
AVL Tree	log V	log V	log V	**O(E log V)**
d-way Heap	$d\log_d V$	$d\log_d V$	$\log_d V$	**$O(E\log_{E/V} V)$**
Fibonacci Heap	1	log V	1	**O(E + V log V)**

Directed Acyclic Graphs (DAG)

Description: A directed graph with no cycles.
There is a **Topological ordering ⇔ Graph is a DAG**. (i.e Every DAG has a Topological ordering)

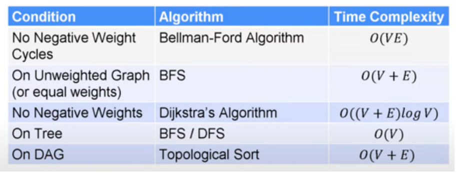
Topological Ordering

Properties:
1 - Sequential total ordering of all nodes
2 - Edges only point forward
3 – Typically Topological ordering are not unique (nodes that are in a different connect component have no dependencies on each other).
Note: Topological ordering is **unique** only if **all adjacent nodes are directly connected to each other.**
Algorithm: Post-order DFS {O(V + E)}
(i.e recurse through all nodes first then process current node)
Kahn's Algorithm:
Repeat:
- S = nodes in G that have no incoming edges.
- Add nodes in S to the topo-order
- Remove all edges adjacent to nodes in S
- Remove nodes in S from the graph

Runtime Complexity: O(V + E)

SSSP for DAG

Idea: TopoSort then relax in order.
Runtime: O(E)
Special property:
- Able to find **longest path** by negating weights / modify relax function (take max instead of min)!



Condition	Algorithm	Time Complexity
No Negative Weight Cycles	Bellman-Ford Algorithm	$O(VE)$
On Unweighted Graph (or equal weights)	BFS	$O(V + E)$
No Negative Weights	Dijkstra's Algorithm	$O((V + E)log\,V)$
On Tree	BFS / DFS	$O(V)$
On DAG	Topological Sort	$O(V + E)$

Priority Queues

Description: Maintains a set of prioritized objects.
Operations:
- **Insert**()
{Sorted Array: O(n), Unsorted Array: O(1), AVL tree: O(logn)}
- **extractMin**() / **extractMax**()
{Sorted Array: O(n), Unsorted Array: O(n), AVL tree: O(logn)}
Removes key with minimum/maximum priority
- **decrease/increaseKey()**
- **contains**(k)
- **isEmpty**()

Uses: Scheduling

Heaps

Binary/MaxHeap

Description: implements a max priority queue that stores the highest priority items at the root and the smallest in the leaves
Does so by mapping each node to a slot in an array.
Properties:
1) Heap ordering: priority[parent] >= priority[child]
(Note: **NOT A BST**, searching takes O(n) time!)
2) Complete Binary Tree: Each level is full and flushed to the left.
Makes Heaps very efficient since the operations comes very close to exactly O(logn) time.
Max height: floor(logn),
Heap Operations: {O(logn)}
- bubble()
checking and swapping nodes to achieve correct priority order in a direction (either up or down)
- insert
add new leaf, then bubble up
- increaseKey
Update priority and bubble up
- decreaseKey
Update priority and bubble down (Choose node with higher priority)
- delete
Swap node with least prioritized node (most right node in the last level)
Delete the node
Bubble down/up the least prioritized node
- extractMax
return root then delete root.

Pros: Same cost for all operations and slightly simpler (no rotations) and better concurrency

Heap sort

Description: Unsorted array -> Heap -> Sorted Array
node.left = 2x + 1
node.right = 2x + 2
parent = floor((x-1) / 2)
Algorithm:
for (n – 1) times we will extractMax and put the result into the last slot of the array.
Invariant: At end of j iteration, the j most prioritized elements extracted will be sorted in final j positions of the array
Runtime Complexity: Always O(nlogn) ->
O(n) to build heap, then extractMax {O(logn)} n times
(Slightly faster than merge sort and slower than quick sort, worse case sorted array)
Stability: Unstable
Space Complexity: O(1)

Disjoint Sets Data Structure (Union-Find)

Dynamic Connectivity
Union: connect two objects
Find: is there a path connecting the two objects?
Transitive: If p is connected to q and if q is connected to r, then p is connected to r.
Note: If object stored are not integers, we will use HashTable with open addressing.

Quick Find

Description: Have 2 int[] consisting of object and component identifier to identify groups of data. Generates FlatTrees

Operations:

Find – return componentId[p] == componentId[q] **{O(1)}**

Union - iterates through the array and updates component identifier **{O(n)}**

Quick Union

Description: Have 2 int[] storing objects and parents respectively.

Two objects are connected if they have the same parent (i.e same root node).

Constructs tall unbalanced trees.

Operations:

Find – walk up the tree and find root node of both objects and compare them. **{O(n)}**

Union – walks up both trees to find root node, set one of the root node parents to be the other root node. **{O(n)}**

Weighted Union

Description: Improved quick union, makes better choice of choosing which root node becomes the parent. Creates a relatively flat tree.

Operation:

Find – same as quick union, walk up the tree and compare root nodes. **{O(logn)}**

Union – When connecting 2 disjoint sets, we will always connect the smaller to the bigger set (i.e we will set the root node of the smaller set to be the root node of the bigger set). **{O(logn)}**

Note: Size of smaller set always doubles.

Maximum depth of tree: **O(logn)**

Path Compression

Description: After finding the root, set the parent of each traversed node to the root.

Runtime Complexity:

Find: O(logn), Union: O(logn) (Expected)

Worse Case: O(n) when we perform series of union in succession then calling find after all the unions.

Weighted Union with Path Compression

Runtime complexity:

Find: O(n + mα(m, n)) = O(α(m, n)), Union: O(α(m, n))

(In any practical application (up to 2^{65533}), this is as good as O(1) time)

Further optimization: If we are dealing with sequences instead of just random sets, and union will only ever be performed on 2 adjacent sequence, we can **further reduce the union time to O(1)** by storing the root node using the right bound of the sequence.

	find	union
quick-find	O(1)	O(n)
quick-union	O(n)	O(n)
weighted-union	O(log n)	O(log n)
path compression	O(log n)	O(log n)
weighted-union with path-compression	α(m, n)	α(m, n)

Minimum Spanning Tree (MSTs)

Properties:

- No Cycles
- If you cut an MST, the two pieces are both MSTs
- For every cycle, the maximum weight edge is not in MST
- For every partition of the nodes, the minimum weight edge across the cut is in the MST

Note: MST does not give shortest path, MST are unique ONLY IF edges are unique

Common misconceptions:

1 - For every cycle, the minimum weight edge MAY OR MAY NOT be in the MST.

2 - For every vertex, the maximum outgoing edge MAY OR MAY NOT be part of the MST.

3 – Joining 2 MST together with shortest edge it MAY NOT give you an MST.

Prim's Algorithm

Description: Maintain a set of visited nodes. Greedily grow the set by adding node connected via lightest edge

Data structure: Priority Queue to order nodes by edge weight

Runtime complexity: O((V+E)logV) => O(ElogV) (Each node inserted/extracted once => O(VlogV), each edge decreaseKey once => O(ElogV))

Kruskal's Algorithm

Description: Sort the edges by weight in ascending order. If the endpoints are in the same component, mark edge red (not in MST), else marked blue(in MST)

Data Structure: Union-Find to check if connected and to connect 2 nodes if they are in the same blue tree.

Runtime complexity : O(Elog V) (sorting), O(α(n)) for each edge.

Boruvka's Algorithm

Description: Add the minimum adjacent edge ("obvious edges") of every node/connected component. Then merge connected components (1 Boruka's Step) (Each iteration k/2 components merge)

Runtime complexity : Total: O(ElogV)

Initial step: O(V) (store component identifier for every node)

"Boruvka's step": O(V + E) BFS/DFS, Merging: O(V). Since max logV "Boruvka's" steps, O((E + V)logV

MST Variants

Constant weight edges

Approach: Get spanning tree (V- 1 edges) with DFS / BFS **{O(E)}**

Bounded integer edge weights (n between 1 to 10)

Approach: (Modified Prim's algo) Use size 10 array of linked lists as PQ. Insert node in correct list, remove by looking up node(i.e using hash table). Extract min by removing from min bucket, and decreaseKey by looking up node and moving it to the correct bucket. **{O(V+E) = O(E)}**

Directed Acyclic Graphs with 1 root

Approach: For every node except roots, add mininmm weight incoming edge **{O(E)}**

Maximum Spanning Tree

Approach: Since adding or multiplying constant to each edge weight does not affect MST, multiply each edge weight by - 1, run MST algo, most "negartve" MST is the maximum spanning tree **{O(ElogV)}**

Steiner Tree Problem (MST of a subset of vertices)

Note: NP Hard, no efficient polynomial time algorithm, however there exist a 2*OPT approximation algorithm.

Approximation approach: (Does not return optimal result!)

For every pair of required vertex, calculate shortest path **O(ElogV)** Construct a new graph based on the shortest paths, then run MST algo on new graph. Finally, map the MST edges back to the original graph

Dynamic Programming

Properties:

1 - Optimal sub-structures: Optimal solution can be constructed from optimal solutions to smaller sub-problems.

2 - Overlapping sub-problems: The same smaller problem is used to solve multiple different bigger problems.

Basic Strategies:

- Bottom-up - Solve the smallest problems (Start with problems with no dependencies) and combine them in order to solve the root problem. (DAG + topological sort then solve in reverse order)
- Top-down - Start from the root and recurse till smaller problem, solve and memoize so each solution is only computed once

Longest Increasing Subsequence

Idea: Start with the smallest problem (node at the end) then for all other nodes, we will examine each outgoing edge, find the maximum and add 1.

Sub-Problem: S[i] = LIS(A[i…n]) starting at A[i] (n subproblems)

Solving: S[n] = 0 (base case, start from the back!)

S[i] = ($\max_{(j < i, A[j] < A[i])}$ S[j]) + 1. (Look at all preceding problems and seeing if we can extend it)

Runtime: O(n^2) (n subproblems, each costing O(i) time)

Bounded Prize Collecting

Aim: To achieve the maximum prize in k steps

Sub-structure: Maximum price in k steps (kV subproblems)

Approach 1: DAG / topological sort. Create k copies of each node and connect each batch of nodes with edges. Create a super source and run SSSP once to get longest path {O(kE)}

Approach 2: P[v, k] = maximum prize that you can collect starting at v and taking exactly k steps. Solve using subproblems P[v, k] = max(P[w1, k-1] + w(v, w1), P[w2, k-1] + w(v, w2), …). Store all P[v,k] in 2x2 matrix for memorization.

Runtime: O(kE) (k rows each needing O(E) time to solve)

Vertex Cover on a **Tree**

To output a set of nodes C of an original tree where every edge is adjacent to at least one node in C

Sub-structure: 2 sub-problems (2V subproblems)

Assuming a subtree with one root v and two children,

- S[v,O] = size of vertex cover in subtree if v is not covered
- S[v,1] = size of vertex cover in subtree if v is covered

Approach : Solve the smallest problems (ie S[leaf, 0] = 0, S[leaf, 1] = 1) While moving up, solve by getting min of two sub-problems. **Intuition**: If root is covered, the children has a choice to be covered or not, hence take min. If parents is not covered, children has to be covered.

Runtime: {O(V)}

All Pair's Shortest Path (Floyd Warshall)

Input: Weighted, directed graph G = (V, E)

Output: dist[v,w]: shortest distance from v to w for all pair of vertices (v,w).

Approach 1: Run Dijkstra once for every vertexv in the graph, assuming weights are all positive

- Sparse graph where E = O(V): O(V^2logV) (Best known solution)
- Unweighted graph use BFS: O(V(E + V)). Dense: O(V^3), Sparse graph: O(V^2)

Sub-structure: If P has the shortest path(u -> v -> w) then P contains the shortest path from (u -> v) and from (v -> w).

Subproblem: S[v,w,P] = the shortest path from v to w that only uses intermediate nodes in the set P (subset of nodes in the graph). (V + 1 possible sets, each set containing nodes 1 though k, i.e P_0 = Ø, P_1 = {1}, P_2 = {1,2} …)

Approach 2: (Dynamic programming)

Base case: S[v,w, Ø] = E[v,w] which is the weight of the edge from directly connecting v to w, else infinity if there is no edge from v to w.

Using subpproblem: S[v,w,P_k] = min(S[v, w, P_{k-1}], S[v, k, P_{k-1}] + S[k, w, P_{k-1}]) {O(V^3)}

Note: Use Dijkstra V times when dealing with sparse graph, and Floyd Warshall when dealing with dense graph.

Probability Theory

E[X] = e_1p_1 +e_2p_2 + … + e_np_n

E[A+B] = E[A]+E[B]

Coin Flips

In two coin flips: I expect one heads when flipping an unfair coin:

Pr(heads) = p

Pr(heads) = 1 – p

E[X] = (p)(1) + (1 – p) (1 + E[X]) => E[X] = 1/p

Simple recurrences

(Hint: to solve recurrences, can guess and verify, draw recursion tree or use Master Theorem, solve by induction)

T(n) = T(n/2) + 1 {O(logn)}

T(n) = 2T(n/2) + 1 {(O(n))

T(n) = T(n/2) + n {O(n)}

T(n) = √nT(√n)+ √n {O(n)} (Sieve of Eratosthenes)

T(n) = 2T(n/2) + n {O(nlogn)}

T(n) = 2T(n/2) + O(n log n) {O(n$\log^2$n)}

T(n) = 1 + T(n-1) + T(n-2) / T(n-1) + T(n-2)

{O(Φ^n) (tight bound) / O(2^n) (loose bound)}

T(n) = T(n/2) + n^2 {O(n^2)}

Sequences/Math Stuff

1+2+3+…+n = n(n+1)/2 = O(n^2)

logN! = O(NlogN) (Sterling's approximation)

n+n/2+n/4+…+n/$2^{\log n}$ = 2n

1+1/2+1/3+1/4+…1/n = O(logn) (Harmonic series)

1+2+4+…+n = O(2n-1)

1+2^2+3^2+…+n^2 = (n/6)(n+1)(2n+1) = O(n^3)

log(ab) = loga + logb

log(a/b) = loga - logb

h(n) = $f(n)^{g(n)}$ in this case **note the constant in g(n)!**

e.g h(n) = $n^{\log n}$ ≠ $n^{2\log n}$

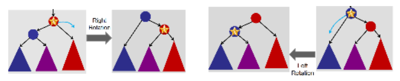
AVL Tree

At most height of 2logn, at least $2^{h/2}$ nodes, at most 2^{h+1} - 1 nodes

After insertion if tree is left heavy:

- Left subtree is **balanced**: Right-rotate
- Left subtree is **left heavy**: Right-rotate
- Left subtree is **right heavy**: Left-rotate then right rotate
- Opposite for right heavy situation

(**Worst case**: 2 rotations)



Rebuilding BST

- InOrder + Pre/Post/Level-Order
- Preorder

Checking AVL: Do post order traversal to correctly calculate height and then use that height to ensure height property is kept.

Hashing

Chaining

Exepected worst case search time: O(logn)

Expected search time = 1 + n/m {O(1)}

Expected max cost of inserting n items = O(logn)

(m = number of buckets, n = number of elements)

Open Addressing

Expected cost of operations = 1 / (1- α), where α = n / m

Sorting

sort	best	average	worst	stable?	memory
bubble	$\Omega(n)$	$O(n^2)$	$O(n^2)$	✓	$O(1)$
selection	$\Omega(n^2)$	$O(n^2)$	$O(n^2)$	×	$O(1)$
insertion	$\Omega(n)$	$O(n^2)$	$O(n^2)$	✓	$O(1)$
merge	$\Omega(n\log n)$	$O(n\log n)$	$O(n\log n)$	✓	$O(n)$
quick	$\Omega(n\log n)$	$O(n\log n)$	$O(n^2)$	×	$O(1)$
heap	$\Omega(n\log n)$	$O(n\log n)$	$O(n\log n)$	×	$O(n)$

Augmented Trees

Order statistics trees (Augmented AVL tree)

Augment: Store weight (number of children) of subtree in each node

Purpose: Finds rank and kth smallest key in O(logn) time

Intervals trees (Augmented AVL tree)

Augment: Store interval in each node to and the max endpoint of subtrees.

Purpose: Find an interval that contains k **(O(logn))** and Finds ALL intervals that contains k. (O(jlogn) where j is the number of overlapping intervals that contains k)

Tournament Tree (Augmented BST)

Augment: Start with perfectly balanced BST with each contender at leaf nodes.

Purpose: Provides a quick way(least comparisons needed) to determine a winner among n contenders. **{O(nlogn)}**

Range trees (Augmented Binary Search Tree)

1-dimensional:

Augment: Store all the **points at leaves** (internal nodes store only copies) and each internal node stores the **max of any leaf in the left subtree**

{Building the tree: O(nlogn), Space: O(n)}

Purpose: Efficient range queries (important for databases) in O(k + logn) time (where k is the number of items found and log n is time to find split node)

2-dimensional:

Augment: Build a x-tree using only x coordinates. For every node in the x-tree, build a y-tree out of nodes in subtree using only y-coordinates

{Building tree: O(nlogn), Space: O(nlogn),

Rotation : O(n)}

Purpose: Efficient range query in 2d space.

{O($\log^2$n + k)}

Miscallaneous

Converting BST to a AVL Tree

Runtime: O(n)

Algorithm: Perform In-Order traversal of the BST which produces a sorted array then run algo for generating AVL Tree from sorted array.

Merging BST/AVL Trees

Runtime: O(M+N) where M and N are size of AVL Trees

Algorithm: Perform In order traversal of the 2 BST which gives 2 sorted Array. Merge the 2 sorted array using merge step of merge sort. Rebuild new AVL Tree using AVL Tree from sorted array Algo.

Generating a Heap from an unsorted Array

Runtime O(n)

Algorithm: start of with all leave nodes as heaps, then we recurse using the fact that left + right are already heaps.

i.e for n-1 times we will call bubbleDown()

Sorted Array to Min/MaxHeap

It is already a heap structure in the form of an array, just return the array!

Runtime: O(1)

Generating an AVL Tree from a sorted array

Runtime cost: O(n)

Algorithm: repeatedly take the middle element to be the root of each iteration.

Union-Find data structure with sequences

- if we are dealing with a contiguous set of tasks instead of random sets then we can use the bounds of the sequence to point to root node directly, reducing run time of union to be O(1).

Joining 2 MST together

Algorithm: Run DFS/BFS to label the 2 connected components. {O(V+E)}. Go through all edges in the original graph and find the smallest edge that links 2 vertices that belong to 2 different components and is not the removed edge. {O(V+E) if using Adjacency List}

Runtime: O(V+E)

MiniMax Path

Solution	Time Complexity		Space Complexity		
	Preprocess	Query	Preprocess	Query	DS
Binary search on query-time trimmed graph	0	$O((V+E)\log k)$	0	$O(V)$	N/A
Binary search on preprocessed trimmed graphs	$O(k(V+E))$	$O(\log k)$	$O(kV)$	$O(1)$	$O(kV)$
Modified relax SSSP with Dijkstra's	0	$O((V+E)\log V)$	0	$O(V)$	N/A
Modified relax APSP with Dijkstra's	$O(V(V+E)\log V)$	$O(1)$	$O(V^2)$	$O(1)$	$O(V^2)$
Modified relax APSP with Floyd-Warshall's	$O(V^3)$	$O(1)$	$O(V^2)$	$O(1)$	$O(V^2)$
Query-time MST†	0	$O(E\log V)$	0	$O(V)$	N/A
Preprocessed MST†	$O(E\log V)$	$O(V)$	$O(V)$	$O(V)$	$O(V)$
Preprocessed MST†+Binary search+LCA	$O(E\log V+V^2)$	$O(\log^2 V)$	$O(V\log V)$	$O(1)$	$O(V\log V)$

†: Implemented using Prim's with binary heap