		A:O(log n)	B:O(n)	C:O(n log n)	D:O(n^2)	E: None
1	`void f(int n) { for (int i = n/4; i < n*2; i+=2) doOne(); }`	○	●	○	○	○
2	`void f(int n) { for (int i = 0; i < n*2; i+=2) for (int j = 0; j < n*2; j++) doOne(); }`	○	○	○	●	○
3	`int f(int n) { if (n < 10) return 0; for (int i = 0; i < n; i++) doOne(); for (int i = 0; i < n; i++) doOne(); f(n-1); }`	○	○	○	●	○
4	`void f(int n) { if (n < 10) return; for (int i = 0; i < n/2; i++) doOne(); f(n/2); f(n/2); }`	○	○	●	○	○
5	`int f(int n) { if (n < 10) return 0; for (int i = 0; i < n; i++) return f(n/2); }` for return statement, hence for loop has no purpose	●	○	○	○	○
6	`int f(int n) { if (n < 10) return 0; for (int i = 0; i < n*n; i+=n) doOne(); f(n/2); }`	○	●	○	○	○

In-place: • No extra space is needed

Merge sort: • O(n) space • Memory used for temp. array

Algorithm	Type	Recursive	Stable	In-Place	Time Complexity		
					Random	Sorted	
						Ascending	Descending
Bubble Sort	Comparison		√	√	$O(N^2)$	$O(N)$	$O(N^2)$
Selection Sort	Comparison			√	$O(N^2)$	$O(N^2)$	$O(N^2)$
Insertion Sort	Comparison		√	√	$O(N^2)$	$O(N)$	$O(N^2)$
Merge Sort	Comparison	√	√		$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
Quick Sort	Comparison	√		√	$O(N \log N)$	$O(N^2)$	$O(N^2)$
Random Quick Sort	Comparison	√			$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
Counting Sort	Non-Comparison		√		$O(N)$	$O(N)$	$O(N)$
Radix Sort	Non-Comparison		√		$O(N)$	$O(N)$	$O(N)$
Paranoid Quick Sort		No	n log n	n log n			

Still need extra O(log n) memory for the call stack

Pivot: $\frac{n}{10} < p < \frac{9n}{10}$

Algorithm in the form of $O\left(1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{N}\right)$ O(log N)

Algorithm in the form of $O\left(N+\frac{N}{2}+\frac{N}{4}+\cdots+\frac{N}{N}\right)$ O(N)

Binary Search Tree (AVL)

Search(v) = O(H) = O(logn)
FindMin/FindMax = O(H) = O(logn)
Successor/Predecessor = O(H) = Ologn
- if v has right/left subtree, find min/max on the subtree as the successor /predecessor
- if not, traverse the ancestors until first vertex w greater/smaller than it
- if still no, v is the max/min of BST

Inorder Traversal O(n)
- Left - root - right
Preorder Traversal O(n)
- root - left - right
Postorder Traversal O(n)
- left - right - root

```
int f(int n) {
    for(int i=1;i<(n/2);i++)
        for(double j=0;j<100;j += (100.0/n))
            doSomething(n/2);
    return 0
}
```
Time complexity of f(n) = O(n²)

```
void f(int n) {
    doOh(n);
    if(n<1) return;
    for(int i=0; i<2; i++)
        f(n/2);
}
```
Time complexity = O(n log n)

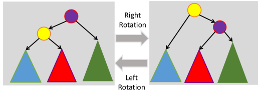
```
int f(int n) {
    if (n<100000)
        return 0;
    for (int i=0;i<(n*n);i++)
        return f(n-1);
}
```
Time complexity of f(n) = O(n)

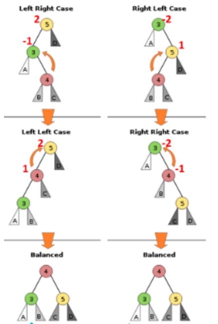
```
int f(n) {
    for(int i=1;i<n;i++)
        for(int j=0; j<i; j+=i)
            doOhOne();
}
```
Time complexity = O(n)

```
void f(double n) {
    if (n<1) return 0;
    doSomething(n);
    for (int i=0;i<10;i++)
        f(n/10);
}
```
Time complexity of f(n) = O(n log n)

```
void f(n) {
    if(n<1) return;
    g(n-1);
}

void g(n) {
    f(n/2)
    doOhOne();
}
```
Time complexity of f(n) = O(log n)





If v left heavy
- If v.left right heavy
  - leftRotate(v.left)
- rightRotate(v)

If v right heavy
- If v.left left heavy
  - rightRotate(v.left)
- leftRotate(v)

AVL Tree Balancing
- Insertion: 0-2 rotations
- Deletion: Up to O(log n) rotations

Insert and remove in O(logN)

Hashing

Direct Addressing Table
- Use a boolean array

1. Search(v): Check if **A[v]** is true (filled) or false (empty),
2. Insert(v): Set **A[v]** to be true (filled),
3. Remove(v): Set **A[v]** to be false (empty).

Operators	Implementations
`search(v)`	Check if A[h(v)] != −1 (where −1 represents empty cell)
`insert(v)`	Set A[h(v)] = v
`remove(v)`	Set A[v] = −1

Load Factor a = N/M
Hash function of h(v) = v%M where m is the size of the has table

A good hash function has the following desirable properties:
a. Fast to compute i.e. $O(1)$.
b. Uses as minimum Hash Table of size M as possible.
c. Scatter the keys into different base addresses as possible.
d. Experience minimum collisions as possible.

Perfect hash function has the following properties:
a. One-to-one mapping between keys and hash values i.e. no collisions at all
b. Table size is the same as the same as the number of keywords supplied.

Chaining

Searching:
- Expected search time = $1 + n/m = O(1)$
- Worst-case search time = $O(n)$

Inserting:
- Worst-case insertion time = O(1)

Division Method
- $h(k) = k \bmod m$
- Choose m = prime number
- Division method is popular (and easy), but not always the most effective.
  - Division is slow.

Multiplication

$$h(k) = \underbrace{(Ak) \bmod 2^w}_{\text{This}} \underbrace{\gg}_{\text{shifted by}} \underbrace{(w-r)}_{\text{this many bits}}$$

- Faster than Division Method
  - Multiplication, shifting faster than division

- Fix table size: $m = 2^r$, for some constant r.
- Fix word size: w, size of a key in bits.
- Fix (odd) constant A.

Linear Probing $(h(F) + f(i)) \bmod m$
f(i) = i

Quadratic Probing $(h(F) + i^2) \bmod m$

Double hashing $h(\text{key},i) = h(\text{key}) + i \times g(\text{key})$

- Set the slot to be "DELETED"

Expected Cost of Action $\leq \frac{1}{1-\alpha}$

Open Addressing Advantages

- Saves space
  - Empty slots vs. linked lists.
- Rarely allocate memory
  - No new list-node allocations.
- Better cache performance
  - Table all in one place in memory
  - Fewer accesses to bring table into cache.
  - Linked lists can wander all over the memory.

How fast should we grow?

- Idea 2: Double the size of the table

  if $(n == m_1)$: $m_2 := 2m_1$

- Assuming n is very large
  - resizing occurs when n was
    - $n/2, n/4, n/8, \ldots$
  - Total time complexity =

    $O(1 + \ldots + n/16 + n/8 + n/4 + n/2 + n) = O(n)$
  - In average, every addition of an item cost O(1)

- Try 2:
  - if $(n == m_1)$: $m_2 := 2m_1$
  - if $(n < m_1/4)$: $m_2 := m_1/2$

Heap

Complete Heap: Every level except last is filled and vertices in last level are as far left
Binary Max Heap: Parent of each vertex contains value greater than or equal to the value of that vertex
Priority Queue ADT
Enqueue(x): put a new element into the PQ
y= Dequeue(): Return an existing y that has the highest priority (key) in the PQ and if ties, return any
Height of tree will not be taller than O(logN) where N is the number of elements

1. **Create(A)** - O(**N** log **N**) version (**N** calls of **Insert(v)** below)
2. **Create(A)** - O(**N**) version
3. **Insert(v)** in O(log **N**) — you are allowed to insert duplicates
4. 3 versions of **ExtractMax()**:
   1. Once, in O(log **N**)
   2. **K** times, i.e., **PartialSort()**, in O(**K** log **N**), or
   3. **N** times, i.e., **HeapSort()**, in O(**N** log **N**)
5. **UpdateKey(i, newv)** in O(log **N** if **i** is known)
6. **Delete(i)** in O(log **N** if **i** is known)

Store heap in array: ▶ left(x), right(x): 2x+1, 2x+2
▶ parent(x): floor$\left(\frac{x-1}{2}\right)$

Operator	Function
`insert(v)`	Insert new item v into the Binary Max Heap at last index $N+1$ to maintain a compact array Fixes the max heap property from insertion point upwards(an operation known as Shift Up/Bubble Up/Increase Key)
`ExtractMax()`	Removes the max item (root) from the Binary Max Heap Last element will replace the root to maintain a compact array Fixes the max heap property from root downwards (an operation known as Shift Down/Bubble Down/Heapify)

$$O(N)$$

Derivation:

$$\text{Height of Binary Tree}, H = \log_2 N$$
$$\text{Number of Nodes at level } h, V = \left\lceil \frac{N}{2^{h+1}} \right\rceil$$
$$\text{Time Complexity of Shift Down of a node at level } h = O(h)$$
$$\therefore \text{Time Complexity of Shift Down at level } h = V \times O(h)$$

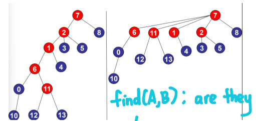
$$\therefore \text{Total Time Complexity}$$
$$= \text{Sum of Time Complexity of Shift Down Operation for all } h$$
$$= \sum_{h=0}^{H} V \times O(h) = O(2N) = O(N)$$

Heapify: Unsorted array to heap

Heapsort faster than MergeSort but slower than quicksort, is **Deterministic** (always nlogn), unstable
Ternary (3-way) heap sort is faster

Heap array to sorted list

Union Find
Initialize O(N)
Findset(i) O(1)
- path compression
IsSameSet(i,j) O(1)
- O(1) as it uses findset
Union O(1)
- link the tree with lower rank to the taller one
- if it is same rank
- resulting tree height increase by one
Path Compression



find(A,B): are they connected?

union(A,B): connect them

	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)

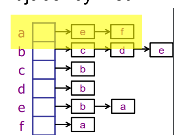
replace component identifier with common ID

replace component identifier with parent ID

Improve quick union. smaller tree added to bigger tree to avoid tall trees

connect each node in the lookup directly to the root (flatten structure of tree)

Adjacency List



Adjacency Matrix

	a	b	c	d	e	f
a	0	0	0	0	1	1
b	0	0	1	1	1	0
c	0	1	0	0	0	0
d	0	1	0	0	0	0
e	1	1	0	0	0	0
f	1	0	0	0	0	0

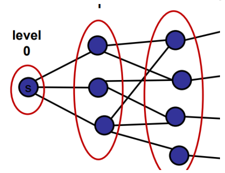
Memory usage for graph G = (V, E):
- Adjacency List: O(V + E)
- Adjacency Matrix: O(V^2)

For a cycle: O(V) vs. O(V^2)
For a clique: O(V + E) = O(V^2) vs. O(V^2)

Base rule: if graph is dense then use an adjacency matrix; else use an adjacency list.

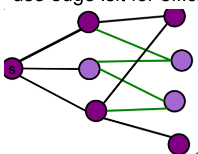
• Degree: No. of adjacent edges.
• Diameter: Shortest path between 2 points

Breadth-First Search O(V + E) Queue



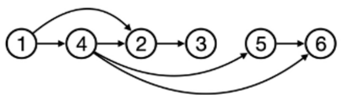
Can be used for SSSP in unweighted graph and Tree
- Each vertex is only visited once as it can only enter the queue once — O(**V**)
- when vertex is dequeued from **queue**, all its **k** neighbors are explored, after all vertices are visited, we have examined all **E** edges — (O(**E**) as the total number of neighbors of each vertex equals to **E**).
- Print traversal path p[v] = u

Depth-first Search O(V + E) Stack
- Used for SSSP with tree as path is unique
- uses another array p[u] to remember the parent of each vertex u
- pick the first neighbor and recursively explore all reachable vertice
- status[u] to check whether it is visited
- Each vertex is only visited once due to the fact that DFS will only recursively explore a vertex **u** if **status[u] = unvisited** — O(**V**)
- when a vertex is visited, all its **k** neighbors are explored, after all vertices are visited, we have examined all **E** edges — (O(**E**) as the total number of neighbors of each vertex equals to **E**)
- use edge lsit for efficiency



• Input: Directed Acyclic Graph
• Output: Total ordering of nodes, where all edges point forward

Topological Sort
- Topological sort of DAG is a linear ordering of DAG vertices in which each vertex comes before all vertices to which it has outbound edges



Algorithm:
→ Post-order depth first search
→ Time: O(V+E)

DFS - add one line of code
- Post-processing by pushing the node into stack

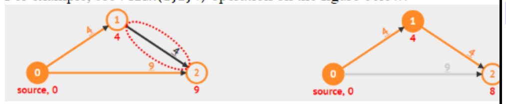
Bipartite Checker
O(V+E) DFS or BFS (they work similarly) to check if a given graph is a Bipartite Graph by giving alternating color (Orange versus blue) between neighboring vertices and report 'non bipartite' if it ends up assigning same color to two adjacent vertices or 'bipartite' if it is possible to do such '2-coloring' process.

Single Source Shortest Path SSSP

Relax

```
relax(u, v, w_u_v)
  if D[v] > D[u]+w_u_v // if the path can be shortened
    D[v] = D[u]+w_u_v // we 'relax' this edge
    p[v] = u // remember/update the predecessor
    // update some other data structure(s) as necessary
```

For example, see relax(1,2,4) operation on the figure below:



Bellman-Ford O(V X E) Slowest

```
for i = 1 to |V|-1 // O(V) here, so O(VxEx1) = O(VxE)
  for each edge(u, v) ∈ E // O(E) here, e.g. by using an Edge List
    relax(u, v, w(u, v)) // O(1) here
```

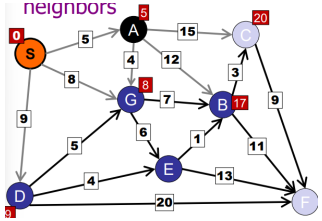
Optimized Bellman-Ford
- Terminate early if no improvement

Detecting -ve Weight Cycle
- Bellman-Ford does not work with -ve weight cycle
- If at least one value **D[u]** fails to converge after |**V**|-1 passes, then there exists a negative-weight cycle reachable from the source vertex **s**.

BFS change the code to use for SSSP
if (visited[v] = 0) { visited[v] = 1 ... } // v is unvisited to
if (D[v] = 1e9) { D[v] = D[u]+1 ... }

Dijkstra Algorithm O((V+E) log V) = O(E log V)
- each vertex will be extracted from the priority queue once, since there are V vertices, this will be done O(V) times
- ExtractMin() runs in O(log V) thus in total O(V log V)
- every time a vertex is processed, we relax its neighbors and in total E edges are processed using DecreaseKey() O(logV)
- in total O(E log V)
- **Wrong answer on graph with -ve weight**

neighbors



Vertex	Dist.
G	8
D	9
B	17
C	20

– BFS: Take edge from vertex that was discovered **least** recently.
 ↳ Use queue
– DFS: Take edge from vertex that was discovered **most** recently.
 ↳ Use stack
– Dijkstra's: Take edge from vertex that is **closest** to source.
 ↳ Use priority queue

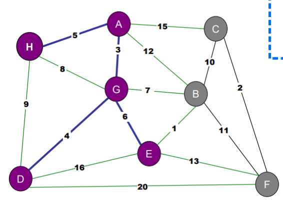
For Acyclic graph, shortest path in negated = longest path in regular with Dijkstra's or using topological sort

Minimum Spanning Tree MST

→ Spanning tree with minimum total weight

Spanning tree is an acyclic subset of edges that connects all nodes
Property 1: No Cycle
Property 2: If you cut an MST, the two pieces are MSTs
Property 3: For every cycle, the max weight edge is not in the MST but the min weight may or may not be in the MST → Instance where edge weights all the same
Property 4: For every partition of the nodes, the minimum weight edge across the cut is in the MST
For every vertex, the min outgoing edge is always part of MST → Basically edge with smallest weight will connect the 2 graphs in the MST

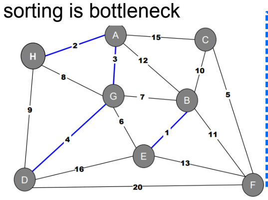
Prim's Algorithm O(E log v) Priority Queue
- needs adjacency List and a boolean array for checking cycles and hashtable for parents
- enqueue all edges incident to s to PQ and greedily extends the tree to include vertex v for the smallest edge if it is not visited until the spanning tree is completed



* Basically Dijkstra's Algorithm

Vertex	Weight
B	1
C	15
F	13

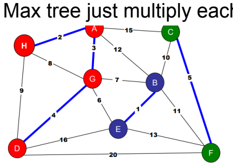
Kruskal's Algorithm O(E log V)
- sort the edges of the graph and use a union-find structure to help in checking the cycle
- loop through the sorted edges and greedily take the next edge e if it does not create any cycle
- sorting is bottleneck



• Basically sort the edges by weights
• If the 2 vertices are already connected, ignore the edge and continue

Boruvka's Algorithm O(E log V) Use UFDS
- add the min outgoing edge of every component
- at most n/2 connected components after every time
- each boruvka step is O(V + E)

If edge is from 1 to 10 weight,we can do counting sort and kruskal's would be faster
MST algorithm does not work on directed edges
We can reweight constantly without affecting paths.
Max tree just multiply each edge by -1



• Create n components
• Connect these components via there minimum outgoing edge
• Repeat process

Convex Hull

Is the collection of all convex combinations

Jarvis March O(hn) O(fn) f is number of faces 3D
- take the leftmost vertex and repeat by choosing the minimal turning angle

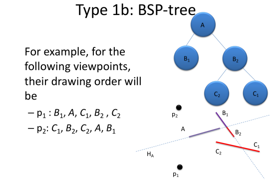
Graham Scan O(nlogn)
- Take left most vertex, sort the rest according to angles and connect according to that order
- if it is convex continue else fall back and check again

Divide and Conquer O(nlogn) O(n^2/nlogn) 3D
- sort vertices in direction and merge every convex hull
- "walk" to the end

Incremental Method
- add a point according to a sorted direction

QuickHull O(n^2) 2D3D
- discard points not in the hull ASAP
- construct a quadrilateral and eliminate points within

Binary Space Partitioning (Using BST)
- Preparation and then rendering

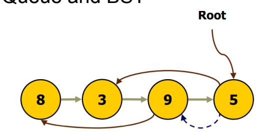


- from the chosen viewpoint render the subtree behind recursively then the node itself then the subtree in front recursively
- Preprocessing may be long and moving environment is tough

Queue and LL (Sorted List head)

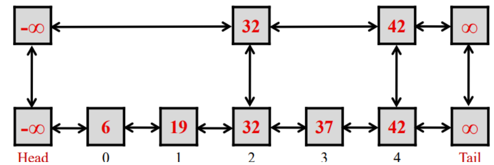
enqueue(item)	O(N)
dequeue()	**O(1)**
peek()	O(1)
printInOrder()	O(N)

Queue and BST



enqueue(item)	O(log N)
dequeue()	**O(1)**
peek()	O(1)
printInOrder()	O(N)

Skip List O(logn)



Planar graph

$v - e + f = 2$ v:vertices e:edges f:faces