MIA - 24 GroupID: C2

I. Searching

- Binary search < cond: array is sorted; running time: $O(\log n)$

code:
```
start = 0
end = len(arr) - 1
while (start < end)
    mid = start + (end - start)/2
    if target <= arr[mid]
        end = mid
    else start = mid + 1
```

Key Idea: Reduce & Conquer (maintain invariant)

- Newton's method $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$
  - usage: find local min. (i.e. $f'(x) = 0$) → needs 2^{nd} derivative
  - ⊕: fast convergence, simple
  - ⊖: slow in higher dimension, $O(d^c)$
- Gradient Descent: more iteration than Newton's but faster comp
  - find good enough γ s.t. $x_{i+1} = x_i - \gamma \cdot f(x_i)$

II. Sorting

Sorting	Worst case	Best case	In-place	stable
Selection (smallest i item in place)	$O(n^2)$	$\theta(n^2)$	✓	✗
Insertion (first i item sorted)	$O(n^2)$	$\theta(n)$ (almost sorted)	✓	✓
Bubble (largest i item in place)	$O(n^2)$ (reverse sort)	$\theta(n)$ (already sorted)	✓	✓
Merge	$O(n \log n)$	$\theta(n \log n)$	$O(n)$	✓
Quick	$O(n^2)$ → identical elements	$\theta(n \log n)$	$O(\log n)$	✗
Radix (i lsn digit sorted)	$O(n)$	$\theta(n)$ $O(d(n+k))$ longest string	$O(n+k)$ possible value	✓
Heap	$O(n \log n)$	$O(n \log n)$	✓	✗

Partitioning

1) Hoare: pivot = 3
[4|1|5|2|3] ↔ [3|1|5|2|4] ↔ [3|1|2|5|4] ↔ [1|2|3|5|4]
lo, hi; lo lo hi; hi,lo

NOT STABLE.

2) Lomuto: pivot = 3
i is first el > pivot
[4|1|5|2|3] → [1|4|5|2|3] → [1|2|5|4|3] → [1|2|3|4|5]

Quick sort optimization: → Stop when < 1024, do insertion sort (almost sorted)
→ pick good pivots (expect 2 tries)

Order Statistics: pick k^{th} largest element.
↳ Quick Select: use partition, go either left/right. $O(n)$.

Knuth shuffle
```
for i = 2 → n do
    r = rand(1, i)
    swap(A, i, r).
```

Sorting shuffle → assign rand to each and sort based on the rand numbers generated.

III. Trees (search trees) → supports $\boxed{\text{Insert, delete, search, successor, predecessor, contains}}$ $O(h)$.
4^n shape of BT; searchMin, searchMax.

1) $\boxed{\text{AVL Tree}}$: max height $2\log(n)$
↳ height-balanced (differ by 1)
↳ Rotation: left-left, left-right, right-right, right-left
↳ $\boxed{O(1)}$

Rotation aft Insertion ≤ 2,
Rotation aft deletion: $O(\log n)$

Traversal: preorder (node, left, right); inorder (left, node, right); postorder (left, right, node) } $O(n)$

construction: $O(n \log n)$, $O(n)$ if sorted → $2T(\frac{n}{2}) + O(1) = T(n)$

Successor
```
If right != null → right.searchMin()
else while (parent != null && child == parent.right) → child = parent
                                                       parent = child.parent
return parent.
```

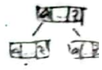
Deletion → 0: just delete; 1: connect child w/ parent
2: update x w/ successor(x). → x = successor(x), delete(x), x.left = x.left, x.right = x.right, parent = x.parent

2) (a,b)-tree / B-tree (b,2b)-tree: More than 1 key

1. root node $2 \le |child| \le b$, internal node $a \le |child| \le b$
2. keys are in sorted order. max height: $O(\log n)$, min height: $\log n$
3. All leaf nodes are in the same depth.

↳ split: lifting median up to parent (insertion) → if w/ contain b keys; ↳ if w/ contain ≤ b keys
(deletion) ↳ merge: Merge siblings if < b el together
↳ share: Merge + split [b, 2b] el together } between siblings

3) Tries: → for strings search: $O(L)$ ← max length of word
→ each node store 1 character / EOW character



4) kd-Trees → split based on $x_1, x_2, \dots, x_k$ cyclically.
↳ search: $O(\sqrt{n})$, construction: $O(n \log n)$ → Quickselect median + partition
$O(kN^{1-\frac{1}{k}})$
can also search nearest neighbor

5) Augmented Trees

i) Order statistics: find k^{th} largest el → $\boxed{\text{EXTRA support}}$ select(k^{th} largest), rank(value) } $O(\log n)$
↳ idea: store weight of subtree

ii) Interval trees: "which of these interval overlap w/ a given interval?" → $O(\log n)$
"which interval contains x?" Search: always go left unless null or max < target.
If all intervals, $O(\log n + k)$
stores intervals (sorted by arrival)
↳ Idea: store max endpoint of subtree
store min also

iii) Range trees: "give me all points in range [x, y]" → $O(\log n + k)$ (go to border; points we take)
store points on leaves!!
↳ idea: store max of left subtree
↳ idea2: add weight of subtree if only need count.
d-dimension: Query: $O(\log^d n + k)$, build: $O(n \log^{d-1} n)$, space: $O(n \log^{d-1} n)$

IV. Hashtables (size M) → supports $\boxed{\text{insert, search, delete, contains}}$ $O(1)$ expected
DO NOT SUPPORT Min, max, successor, predecessor

$\boxed{\text{load}(\alpha): \frac{n}{M}}$

hash function → collision → i) chaining: space: $O(M+n)$
- insert: $O(1)$ worst; delete: $O(1+\alpha)$, $O(n)$ worst; search: $O(n)$ worst, $O(1+\alpha)$ expected
- expected max cost of n insertions, $\alpha = 1$: $\theta(\log n / \log\log n)$

key	value
address →	
→	
→	
→	

HASH Set, Map

ii) Open addressing: → problems: clustering ← cache
a) Linear probing: deletion: set tombstone. ← (i, m) =
b) double hashing: $f(k) + i \cdot g(k) \bmod m$.

Expected cost: $O(\frac{1}{1-\alpha}) \to 1 + \frac{n}{m}(1 + \frac{n-1}{m-1}(1 + \dots$

PROBLEM: SPACE !!
⊕ save space, rarely allocate memory, cache (array w
⊖ sensitive to load, to hash functions.

Resize: double the table. → after $n \ge \frac{M}{2}$ } → amortized $O(1)$
half the table → after $n \le \frac{M}{4}$

differentiator
Fingerprint HT & Bloom: do not store key → only (0,1) vector → lead to $\boxed{\text{false positives}}$

$\boxed{\text{BIG keys are expensive}}$ - false positives < p. Pquery return false ⊕: $1 - (1 - \frac{1}{m})^n \approx 1 - (\frac{1}{e})^{\frac{n}{m}} < p \Leftrightarrow \frac{n}{m} \le \log(\frac{1}{1-p})$
p = 0.1, m ≈ 10n; p = 0.05, m ≈ 20n.

Bloom: 2 hash function.
$(1 - (1 - \frac{1}{m})^{2n})^2 \approx (1 - (\frac{1}{e})^{\frac{2n}{m}})^2 \Leftrightarrow \frac{n}{m} \le \frac{1}{2}\log(\frac{1}{1-\sqrt{p}})$
optimal: $\boxed{\frac{M}{n}\log 2}$ fns
p = 0.1, m ~ 5n; p = 0.05, m ≈ 7n

⊕ Reduced space (no key storing)
⊖ bigger table to avoid collisions

key	
0	1
1	1
2	1
3	0
4	0
5	1
6	0

hash(x), hash(y)

CHAINING (worst/avg)		search	insert	Delete
	Sorted Arr	log n / log n	n / n/2	n / n/2
	Unsorted	n / n/2	n / n	n / n/2
	LL	n / 1	n / 1	n / 1
	[illegible]	n / 1	n / 1	n / 1

Skiplists → search, insert, delete : O(log n) w/ high probability
↳ max level for n elements : O(log n) w/ — "

OpenAddressing : search → hash(key, 0) delete → tombstone
→ hash(key, 1)
hash(key, 2)
CANNOT SEARCH/INSERT AFT FULL

PQ & Binary Heaps

→ usually combined w/ HT to get the desired key

successor/predecessor O(n)

↳ Insert, extractMin, decreaseKey → O(log n)
↳ need to maintain structure

Binary Heaps → complete binary tree (all leaves are to the left). (need to swap w/ left)
→ max height: ⌊log n⌋
→ can handle duplicates
→ build heap in O(n)! → all leaves are heaps itself
so $O(\frac{n}{2}) + O(\frac{n}{4}) + \ldots = O(n)$

Heap sort : extractMax → put at the back → safe because extract max empty last index of array
UNSTABLE , O(n log n) worst, in place

GRAPHS

→ adj. list : O(V+E) space, good for sparse graphs
↳ adj. matrix : O(V²) space, good for dense graphs

Directed
- only outgoing / ingoing edges
- not symmetric

- Traversal → BFS } O(V+E) on adj. list → Queue
↳ DFS } O(V²) on adj. matrix → stack
O(V) space. (stack/queue)

store visited nodes
CANNOT EXPLORE ALL PATHS !! ◇◇◇ exponential

- DAG / Toposort = Post-order DFS : O(V+E)
→ put to back

EACH NODE IS ONLY PROCESSED ONCE
↳ Algo 1 : Post-order DFS (process u iff neighbor[u] have been processed)
↳ Algo 2 : Find u w/o incoming edge , add to front , process edges, (Kahn) remove u and edges adj to u .

Def: **Strongly Connected Components** : ∃ path u→v, v→u . u,v diff comp.

DAG iff ∃ Topo ORDER ← NOT guaranteed UNIQUE tho

SHORTEST PATHS

input	Algo
whatever	BF, O(VE)
unweighted / all same	BFS, O(V+E)
Tree	BFS/DFS, O(V+E)
≥0 edges	Dijkstra O(E log V)
DAGs	Toposort + relax O(V+E)

No Algo work if ⊖ have ⊖ Cycle

① Bellman Ford :
if no ⊖ cycle guaranteed is enough.
↳ V-1 * (relax every edge)
↳ +1 * (relax every edge) → check for ⊖ cycle

STOP early IF |E| relaxation do not change any weight ← can be used to check if an estimate is correct

INVARIANT : After i^{th} iteration of BF
- we have consider every path w/ at most i edges
- ∀ v, distEst[v] ≤ weight of any path from source to v of at most i hops/edges.

② Dijkstra (NONNEGATIVE EDGE ONLY) . Idea: - Relax in correct order s.t every edge only relax ONCE

INVARIANT : all processed vertice estimates is correct .

start w/ node → add shortest path in PQ to the "explored" set

COMPARISON
- Queue : BFS : Take v discovered least recently
- Stack : DFS : most
- PQ . Dijkstra Take v w/ closest dist. to source

BIG IDEA :
- maintain set of "explored" vertices
- add v by following edges explored → not explored

PQ DS	ins	decKey	delMin
Array	1	1	V
AVL	log V	log V	log V
d-way heap	d log_d V	log_d V	d log_d V
Fib heap	1	1	log V

Dijkstra = O(V* (insert + delMin) + E* decKey) → E log V ∀ AVL or heap

Repeat :
- find unexplored v w/ smallest est
- relax all outgoing edges
- mark vertex finished

contains . HT < keys , idx in heap/tree >

NO CYCLE IN SP TREE

- SHORTEST PATH MODS
① Undirected : no ⊖ edge → Dijkstra , if ⊖ edge , BF also cannot (⊖cycle)
② Longest PATH → no ⊕ cycle : ⊖ weight + SSSP .
PROBLEM ⊕ weight cycle
↳ on DAGS → ⊖ all edge → Toposort + relax .

- UFDS < Id array → store parents . obj → just turn into ints by open addressing ; size array .

h : ⌊log n⌋

QF : O(1) find, O(n) union
QU : O(n) find, O(n) union
skewed tree QUPC : O(n) worst case
WQU : O(log n) find , O(log n) union
WQUPC : O(α(n)) find , O(α(n)) union
↑ Ackermann function .

height increases iff 2 trees of same height combined .

MINIMUM SPANNING TREE

≠ SP , use it for minimax / maximin

① No cycle
② Cut an MST → 2 MST .
↳ ∀ cut can have ≥ 1 edge connecting
③ Heaviest edge on cycle ∉ MST (RED Prop)
④ min edge crossing a cut ∈ MST (BLUE Prop)

UNDIRECTED

- PRIM (blue-only strategy)
↳ each added edge is lightest on some cut
O (V* extractmin + E* deckey) = O(E log V) on AVL / heaps

Prim	Dijkstra
edge weight	estimate + edge weight

- KRUSKAL → use UFDS
O(E log E)
sort + O(E* UF operation) ~ O(E log V)
(α or log V)

- Sort
- Repeat :
  - Check whether (u,v) is in same component
  - merge (union) or do nothing

- BORUVKA → use UFDS → parallelizable .
↳ add all dor edges .
1 boruvka step :
O(V+E) - search for min outgoing edge ← iterate thru edgelist and store it in some cheapest
union O(V) { - add to MST
- contract/merge connected comps

Total : log V Boruvka steps
In the beginning, we start with V connected components

- SPECIAL CASES

Input	
unweighted / homogeneous	BFS/DFS O(V+E)
edges ∈ {1,2,...,k}	
directed	
steiner tree	

① Kruskal's variant O(kE) : sort in linear time (bucket sort)
② Prim's variant : O(V+E)
extract min : O(1) → just iterate
deckey : Lazy Deletion
insert : O(1) ← LL just need to store seen nodes in HT

① DAG w/ one root : ∀ node != root, add min incoming e O(V+E)
② MAXST : ⊖ all edge , run MST
NOTE : reweighting edges doesn't matter in MST (± k)

→ Approx algo - ∀ pair, find SP only required
- construct new G' : (V', E') SP(weight)
- build MST
- map back to original graph

sketch proof : DFS on opt
kill steiner nodes all on G'
and subset of its edges .
↳ T is MST on G'

TSP MST : MST → DFS → ignore repeated nodes

- APSP → run SSSP V times < O(V(V+E)) BFS on unweighted .
O(VE log V) Dijkstra from every node
↳ Floyd Warshall
↳ subproblem : A → ? → B .
$P_i = \{1, \ldots, i\}$
|?| = 1, 2, ..., |V| - 2 .
$O(V^3)$

basecase : $S[v,w,\phi] = E[v,w]$
relation : $S[v,w,P_{i+1}] = \min(S[v,w,P_i],\ S[v,i+1,P_i] + S[i+1,w,P_i])$

- class NeighborList extends ArrayList
- class Node { int key; NeighborList neighbors }
- class Graph { Node[] vertices }

DFS (recursive)

```
DFS-Visit (Node[] nodeList, boolean[] visited, int start) {
    for (Integer v : nodeList[start].neighbors) {
        if (!visited[v]) {
            visited [v] = true
            DFS-Visit (nodeList, visited, v);
        }
    }
}

DFS (Node[] nodeList) {
    boolean visited = new boolean[nodeList.length]
    for (start = 0; start < nodeList.length; start++) {
        if (!visited[start]) {
            visited [start] = true;
            DFS-Visit (nodeList, visited, start);
        }
    }
}
```

BFS (recursive)

```
BFS (Node[] nodeList, int startId) {
    boolean[] visited = new boolean[nodeList.length];
    int[] parent = new int [nodeList.length]
    Arrays.fill(parent, -1);
    Collection<Integer> frontier = new Collection<Integer>();
    frontier.add(startId).
    while (!frontier.isEmpty()) {
        Collection<Integer> nextFrontier = new Collection<Integer>();
        for (Integer v : frontier) {
            for (Integer w : nodeList[v].neighbors) {
                if (!visited [w]) {
                    visited [w] = true
                    nextFrontier.add(w)
                    parent [w] = v.
                }
            }
        }
        frontier = nextFrontier
    }
}
```

BFS/DFS (iterative)

```
BFS (Node[] nodeList, int start) {
    Queue<Integer> q = new LL<>();
    q.add(start);
    while (!q.isEmpty()) {
        int curr = q.poll();  (or pop())
        for (Integer v : nodeList[curr].neighbors) {
            q.add(v);
        }
    }
}
```

BF →

```
for (i=0; i < V.length; i++)
    for (Edge e : edgelist)
        relax(e)

relax (int u, int v) {
    if (dist[v] > dist[u] + w(u,v)) {
        dist[v] = dist[u] + w(u,v)
    }   parent[v] = u
}
```

Dijkstra

```
public Dijkstra {
    Graph G,
    PriorityQueue pq;
    double[] distTo;

    searchPath (int start) {
        pq.insert(start, 0)
        distTo = new double[G.size()]
        Arrays.fill(distTo, Infty);
        distTo [start] = 0
        while (!pq.isEmpty()) {   ← V times
            int curr = pq.delMin();
            for (Edge e : G[curr].neighbors) {
                relax(e);
            }
    }

relax (Edge e) {   ← E times
    int v = e.from();
    int w = e.to()
    double weight = e.weight();
    if (distTo[w] > distTo[v] + weight) {
        distTo[w] = distTo[v] + weight;
        parent [w] = v
        if (pq.contains(w)) {
            pq.decKey (w , distTo[w]);
        } else { pq.ins (w, distTo[w]);
    }
}
```

WQU

```
union (int p, int q) {
    int parent p = find(p)
    int parent q = find(q)
    if (size[parent p] > size[parent q])
        parent [parent q] = parent p
        size [parent p] += size [parent q]
    else
        parent [parent p] = parent q
        size [parent q] += size [parent p].
```

PC

```
find (p) {
    root = p;
    while (parent[root] != root) {
        parent[root] = parent[parent[root]]   ← PC
        root = parent[root]
    }
    return root;
}
```

OR

```
find(p) {
    if (parent[p]==p) return p
    parent[p] = find(parent[p])   ← PC
    return parent[p]
}
```

Cycle Detection (Tarjan) → DFS O(V+E)

```
void dfs (int u) {
    isVisited[u] = true
    isInStack[u] = true
    for (int v : adjList[u]) {
        if (isVisited[v] && isInStack[v])
            print ("Cycle Detected")
        if (isVisited[v])
            continue;
        dfs(v)
    }
    isInStack [u] = false
}
```

OR

```
dfs (int v, int parent) {
    color[v] = 1
    for (int w : adjList[v]) {
        if (color[w] == 1) cycle!
        else if (color[w] == 0)
            dfs (w, v)
    }
    color[v] = 2
}
```

PRIM

```
• PriorityQueue pq
  for (Node v : G.V()) {
      pq.add(v, Infty);
  }
  pq.deckey(start, 0)
• HashSet <Node> seen            • int[] currEstimate
  seen.put(start)                  Arrays.fill(currEstimate, Infty)
• HashMap <Node, Node> parent
  parent.put(start, null)
while (pq.isEmpty()) {
    int curr = pq.delMin(); seen.put(curr),
    for (Edge e : curredgeList()) {
        Node w = e.otherNode(curr);
        if (!seen.get(w) && currEstimate[w] > e.getWeight())
            pq.deckey(w, e.getWeight())
            currEstimate[w] = e.getWeight() ?
            parent.put(w, curr);
        }
    }
}
```

Kruskal

```
Edge[] sortedEdges = G.E() sort().
ArrayList<Edge> mstEdges = new ArrayList<Edge>()
UnionFind uf = new UnionFind(G.V())

for (int i = 0; i < sortedEdges.length; i++) {
    Edge e = sortedEdges[i]
    Node v = e.one()
    Node w = e.two()
    if (!uf.find(v) == uf.find(w)) {
        mstEdges.add(e);
        uf.union(v, w);
    }
}
```

Minimax

	Preprocess	Extraspace	Query (s,D)
Dj	–	–	run dj $O(E\log V)$.
	Dj on all v $O(VE\log V)$	store all badness $O(V^2)$	lookup $O(1)$
MST	–	–	MST+DFS $O(E\lg V)$.
	any MST also $O(E\lg V)$	store MST $O(V)$	DFS $O(V)$.
	–	–	$O(E\lg k)$ Bs on k.
Binsev	G' for all k. $O(kE)$	$O(kV)$ store connected components info	$O(\lg k)$

FW $O(V^3)$

```
int[][] APSP(E) {
    int[][] S = new int[V.length][E.length];
    for (int v=0; v<V.length; v++) {
        for (int w=0; w<V.length; w++) {
            S[v][w] = E[v][w];

    for (int k=0; k<V.length; k++) {
        for (int v=0; v<V.length; v++) {
            for (int w=0; w<V.length; w++) {
                S[v][w] = min(S[v][w], S[v][k]+S[k][w]);
```

DFS Topo

```
DFSrec(u, nodeList, visited)

    for (int v : G[u].neighbors) {
        if (!visited[v])
            visited[v] = true
            DFS-Visit(nodeList, visited, v)
            put v at the back of Array.
    }.
}.
```

Khan

```
for all v in V.
    indeg[v] ← 0
    parent[v] ← -1.
for each edge (u,v) in G
    indeg[v]++
for all v where indeg[v]=0
    Q.add(v).

while !Q.isEmpty()
    int curr = Q.delMin()
    for all v in edgeList(curr).
        if indeg[v] > 0
            indeg[v]--;
        if indeg[v] == 0.
            Q.add(v)
            parent[v] = curr

toposort in order
```