1. Arrays (Lists)

Python lists are dynamic arrays.

```
# Access O(1), append O(1) amortized, insert/delete O(n)
arr = [1,2,3]
x = arr[1]                 # 2
arr.append(4)              # [1,2,3,4]
arr.insert(1, 99)          # [1,99,2,3,4]
arr.remove(3)              # O(n)
print(4 in arr)            # membership: O(n)
```

2. Tuples & Sets

Tuples are immutable sequences; sets are hash-based unique collections.

```
t = (1,2,3)                # immutable
s = {1,2,3}
s.add(2)                   # no duplicate
s.update({3,4})
print(2 in s)              # O(1) average
s.remove(4)                # KeyError if missing
```

3. Dictionaries (Hash Maps)

Average O(1) for insert/search/delete.

```
d = {}
d["name"] = "Alice"        # insert O(1)
print(d.get("name"))       # read O(1)
d.pop("name", None)        # delete O(1)
for k, v in d.items():     # iterate O(n)
    pass
```

4. Stack

LIFO using list.

```
stack = []
stack.append(10)           # push O(1)
stack.append(20)
top = stack[-1]            # peek O(1)
val = stack.pop()          # pop O(1)
```

5. Queue & Deque

FIFO queue and double-ended queue.

```
from collections import deque
q = deque()
q.append(1); q.append(2)   # enqueue O(1)
x = q.popleft()            # dequeue O(1)

dq = deque([1,2,3])
dq.appendleft(0)           # O(1)
dq.pop()                   # O(1)
```

6. Singly Linked List

Insert at head O(1), traverse O(n).

```
class Node:
    def __init__(self, data, nxt=None):
        self.data, self.next = data, nxt
class LinkedList:
    def __init__(self): self.head = None
    def insert_head(self, x):
        self.head = Node(x, self.head)
    def search(self, x):
        cur = self.head
        while cur:
            if cur.data == x: return True
            cur = cur.next
        return False
    def __iter__(self):
        cur = self.head
        while cur:
            yield cur.data
            cur = cur.next

ll = LinkedList()
ll.insert_head(3); ll.insert_head(5)
print(list(ll))            # [5,3]
```

7. Binary Tree Traversal

DFS traversals are O(n).

```
class T:
    def __init__(self, v, l=None, r=None):
        self.v, self.l, self.r = v, l, r

def inorder(root):
    if not root: return
    yield from inorder(root.l)
    yield root.v
    yield from inorder(root.r)

root = T(2, T(1), T(3))
print(list(inorder(root)))  # [1,2,3]
```

8. Binary Search

Works on sorted arrays. O(log n).

```
def binary_search(a, target):
    lo, hi = 0, len(a)-1
    while lo <= hi:
        mid = (lo+hi)//2
        if a[mid] == target: return mid
        if a[mid] < target: lo = mid+1
        else: hi = mid-1
    return -1

print(binary_search([1,3,5,7,9], 7))  # 3
```

9. Sorting (Merge & Quick)

MergeSort O(n log n), QuickSort avg O(n log n).

```
def mergesort(a):
    if len(a)<=1: return a
    m = len(a)//2
    L, R = mergesort(a[:m]), mergesort(a[m:])
    i=j=0; out=[]
    while i<len(L) and j<len(R):
        if L[i] <= R[j]: out.append(L[i]); i+=1
        else: out.append(R[j]); j+=1
    out.extend(L[i:]); out.extend(R[j:])
    return out

def quicksort(a):
    if len(a)<=1: return a
    p = a[len(a)//2]
    return quicksort([x for x in a if x<p]) + [x for x in a if x==p] + quicksort([x for x in a if x>

print(mergesort([5,2,4,6,1,3]))
print(quicksort([5,2,4,6,1,3]))
```

10. Graph: BFS & DFS (Adjacency List)

O(V+E).

```
from collections import deque

g = {'A':['B','C'],'B':['D'],'C':['D'],'D':[]}

def bfs(start):
    vis=set([start]); q=deque([start])
    order=[]
    while q:
        u=q.popleft(); order.append(u)
        for v in g[u]:
            if v not in vis:
                vis.add(v); q.append(v)
    return order

def dfs(u, vis=None, order=None):
    if vis is None: vis=set()
    if order is None: order=[]
    vis.add(u); order.append(u)
    for v in g[u]:
        if v not in vis: dfs(v, vis, order)
    return order

print(bfs('A'))   # ['A','B','C','D']
print(dfs('A'))   # ['A','B','D','C']
```

11. Topological Sort (Kahn's Algorithm)

For DAGs. O(V+E).

```
from collections import deque, defaultdict
g = {0:[1,2], 1:[3], 2:[3], 3:[]}
indeg = defaultdict(int)
for u in g:
    for v in g[u]: indeg[v]+=1
q = deque([u for u in g if indeg[u]==0])
order=[]
while q:
    u=q.popleft(); order.append(u)
    for v in g[u]:
        indeg[v]-=1
        if indeg[v]==0: q.append(v)
print(order)  # valid topo order
```

12. Dijkstra (Shortest Paths)

Non-negative weights. O((V+E) log V).

```
import heapq
```

```
def dijkstra(graph, start):
    dist={u: float('inf') for u in graph}; dist[start]=0
    pq=[(0,start)]
    while pq:
        d,u=heapq.heappop(pq)
        if d!=dist[u]: continue
        for v,w in graph[u]:
            nd=d+w
            if nd<dist[v]:
                dist[v]=nd; heapq.heappush(pq,(nd,v))
    return dist

graph = {'A':[('B',1),('C',4)], 'B':[('C',2),('D',5)], 'C':[('D',1)], 'D':[]}
print(dijkstra(graph,'A'))  # {'A':0,'B':1,'C':3,'D':4}
```

13. Bellman-Ford

Handles negative edges; detects negative cycles. O(VE).

```
def bellman_ford(n, edges, src):
    INF=10**15
    dist=[INF]*n; dist[src]=0
    for _ in range(n-1):
        updated=False
        for u,v,w in edges:
            if dist[u]!=INF and dist[u]+w<dist[v]:
                dist[v]=dist[u]+w; updated=True
        if not updated: break
    # detect negative cycle
    for u,v,w in edges:
        if dist[u]!=INF and dist[u]+w<dist[v]:
            raise ValueError("Negative cycle")
    return dist

edges=[(0,1,1),(1,2,2),(0,2,10)]
print(bellman_ford(3, edges, 0))  # [0,1,3]
```

14. Floyd–Warshall

All-pairs shortest paths. O(V^3).

```
def floyd_warshall(dist):
    n=len(dist)
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][k]+dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k]+dist[k][j]
    return dist

INF=10**9
dist=[[0,3,INF],[INF0:=INF,0,1],[INF,INF,0]]
# fix row 1 INF typo:
dist[1][0]=4
print(floyd_warshall(dist))
```

15. Greedy: Activity Selection

Select max non-overlapping intervals. O(n log n).

```
intervals = [(1,3),(2,4),(3,5),(0,6),(5,7),(8,9)]
intervals.sort(key=lambda x: x[1])
res=[]; end=-float('inf')
```

```
for s,e in intervals:
    if s>=end:
        res.append((s,e)); end=e
print(res)
```

16. Backtracking: N-Queens (count)

Place N queens such that none attack each other. ~O(N!).

```
def total_n_queens(n):
    cols=set(); diag1=set(); diag2=set(); ans=0
    def bt(r):
        nonlocal ans
        if r==n: ans+=1; return
        for c in range(n):
            if c in cols or (r-c) in diag1 or (r+c) in diag2: continue
            cols.add(c); diag1.add(r-c); diag2.add(r+c)
            bt(r+1)
            cols.remove(c); diag1.remove(r-c); diag2.remove(r+c)
    bt(0); return ans

print(total_n_queens(4))  # 2
```

17. Disjoint Set Union (Union-Find)

Amortized ~O(α(n)).

```
class DSU:
    def __init__(self,n):
        self.p=list(range(n)); self.r=[0]*n
    def find(self,x):
        if self.p[x]!=x: self.p[x]=self.find(self.p[x])
        return self.p[x]
    def union(self,a,b):
        ra,rb=self.find(a),self.find(b)
        if ra==rb: return False
        if self.r[ra]<self.r[rb]: ra,rb=rb,ra
        self.p[rb]=ra
        if self.r[ra]==self.r[rb]: self.r[ra]+=1
        return True

d=DSU(5); d.union(0,1); d.union(3,4); print(d.find(1)==d.find(0))
```

18. Kruskal's MST

Sort edges; use DSU. O(E log E).

```
def kruskal(n, edges):
    edges.sort(key=lambda x:x[2])
    d=DSU(n); mst=[]; cost=0
    for u,v,w in edges:
        if d.union(u,v):
            mst.append((u,v,w)); cost+=w
    return cost, mst

edges=[(0,1,1),(1,2,2),(0,2,3)]
print(kruskal(3, edges))
```

19. Prim's MST (Heap)

O(E log V).

```
import heapq
```

```
def prim(adj, start=0):
    n=len(adj); vis=[False]*n
    pq=[(0,start,-1)]; total=0; mst=[]
    while pq:
        w,u,par=heapq.heappop(pq)
        if vis[u]: continue
        vis[u]=True; total+=w
        if par!=-1: mst.append((par,u,w))
        for v,wt in adj[u]:
            if not vis[v]: heapq.heappush(pq,(wt,v,u))
    return total, mst

adj=[[(1,1),(2,3)],[(0,1),(2,2)],[(0,3),(1,2)]]
print(prim(adj,0))
```

20. Segment Tree (Range Sum)

Build O(n), query/update O(log n).

```
class SegTree:
    def __init__(self, arr):
        self.n=len(arr)
        self.t=[0]*(4*self.n)
        def build(i,l,r):
            if l==r: self.t[i]=arr[l]; return
            m=(l+r)//2
            build(i*2,l,m); build(i*2+1,m+1,r)
            self.t[i]=self.t[i*2]+self.t[i*2+1]
        build(1,0,self.n-1)
    def query(self, L, R):
        def q(i,l,r):
            if R<l or r<L: return 0
            if L<=l and r<=R: return self.t[i]
            m=(l+r)//2
            return q(i*2,l,m)+q(i*2+1,m+1,r)
        return q(1,0,self.n-1)
    def update(self, idx, val):
        def upd(i,l,r):
            if l==r: self.t[i]=val; return
            m=(l+r)//2
            if idx<=m: upd(i*2,l,m)
            else: upd(i*2+1,m+1,r)
            self.t[i]=self.t[i*2]+self.t[i*2+1]
        upd(1,0,self.n-1)

arr=[1,3,5,7,9,11]; st=SegTree(arr)
print(st.query(1,3)); st.update(1,10); print(st.query(1,3))
```

21. Fenwick Tree (Binary Indexed Tree)

Point update, prefix sum O(log n).

```
class BIT:
    def __init__(self,n):
        self.n=n; self.b=[0]*(n+1)
    def add(self,i,delta):
        i+=1
        while i<=self.n:
            self.b[i]+=delta
            i+=i&-i
    def sum(self,i): # prefix [0..i]
        i+=1; s=0
        while i>0:
            s+=self.b[i]
            i-=i&-i
        return s
    def range_sum(self,l,r): return self.sum(r)-self.sum(l-1)

bit=BIT(5); bit.add(0,1); bit.add(3,4)
print(bit.range_sum(0,3))  # 5
```

22. KMP (Substring Search)

Pattern matching in O(n+m).

```
def kmp_build(p):
    lps=[0]*len(p); j=0
    for i in range(1,len(p)):
        while j>0 and p[i]!=p[j]: j=lps[j-1]
        if p[i]==p[j]: j+=1; lps[i]=j
    return lps

def kmp_search(s,p):
    lps=kmp_build(p); j=0; idx=[]
    for i,ch in enumerate(s):
        while j>0 and ch!=p[j]: j=lps[j-1]
        if ch==p[j]: j+=1
        if j==len(p): idx.append(i-j+1); j=lps[j-1]
    return idx

print(kmp_search("ababcabcabababd","ababd"))  # [10]
```

23. LRU Cache (OrderedDict)

O(1) average get/put.

```
from collections import OrderedDict
class LRU:
    def __init__(self, cap):
        self.cap=cap; self.od=OrderedDict()
    def get(self, k):
        if k not in self.od: return -1
        self.od.move_to_end(k)
        return self.od[k]
    def put(self, k, v):
        if k in self.od:
            self.od.move_to_end(k)
        self.od[k]=v
        if len(self.od)>self.cap:
            self.od.popitem(last=False)

l=LRU(2); l.put(1,1); l.put(2,2); l.get(1); l.put(3,3)
print(l.get(2))  # -1 evicted
```

24. DP: 0/1 Knapsack

O(nW) time and O(W) space.

```
def knapsack(weights, values, W):
    dp=[0]*(W+1)
    for w,v in zip(weights, values):
        for cap in range(W, w-1, -1):
            dp[cap]=max(dp[cap], dp[cap-w]+v)
    return dp[W]
print(knapsack([2,3,4],[4,5,10],5))  # 9
```

25. Sliding Window

Longest substring without repeating chars. O(n).

```
def longest_unique_substring(s):
    seen={}; start=0; best=0
    for i,ch in enumerate(s):
        if ch in seen and seen[ch]>=start:
            start=seen[ch]+1
        seen[ch]=i
        best=max(best, i-start+1)
    return best

print(longest_unique_substring("abcabcbb"))  # 3
```

26. Two Pointers

Find pair sum in sorted array. O(n).

```
def two_sum_sorted(a, target):
    i, j = 0, len(a)-1
    while i<j:
        s=a[i]+a[j]
        if s==target: return (i,j)
        if s<target: i+=1
        else: j-=1
    return None

print(two_sum_sorted([1,2,3,4,6,8], 10))  # (1,5)
```

Big-O Cheatsheet (Common Ops)

Amortized or average cases unless stated.

```
Array:    access O(1), search O(n), insert/delete middle O(n)
Stack:    push/pop O(1)
Queue:    enqueue/dequeue O(1)
Deque:    append/appendleft/pop/popleft O(1)
Dict/Set: insert/find/delete O(1) avg, O(n) worst
LinkedList: search O(n), insert/delete at head O(1)
Binary Search: O(log n)
Heaps:   push/pop O(log n), find-min O(1)
Graphs:  BFS/DFS O(V+E)
Sorts:   Merge O(n log n), Quick avg O(n log n), worst O(n^2)
Union-Find: almost O(1) (inverse Ackermann)
Segment Tree/Fenwick: query/update O(log n)
Dijkstra: O((V+E) log V) with heap
Bellman-Ford: O(VE), Floyd-Warshall: O(V^3)
Kruskal/Prim: O(E log V)
Backtracking (e.g., N-Queens): exponential
```