ICPC Algorithms Cheatsheet – with Python Templates

Compact, contest-ready Python snippets for >50 core ICPC topics. Mỗi đoạn code là skeleton tối giản, dễ paste & chỉnh nhanh.

1) Cấu trúc dữ liệu & kỹ thuật nền tảng

Prefix sum & Difference array

```
def prefix_sum(a):
    ps = [0]
    for x in a: ps.append(ps[-1]+x)
    return ps  # sum[i..j] = ps[j+1]-ps[i]

def diff_apply(n, updates):  # updates: (l,r,delta) inclusive
    d = [0]*(n+1)
    for l,r,val in updates:
        d[l]+=val
        if r+1<n: d[r+1]-=val
    cur=0; out=[0]*n
    for i in range(n):
        cur+=d[i]; out[i]=cur
    return out
```

Two pointers / Sliding window (sum ≤ K)

```
def longest_subarray_leq_k(a, K):
    s=0; l=0; best=0
    for r,x in enumerate(a):
        s+=x
        while s>K and l<=r:
            s-=a[l]; l+=1
        best=max(best, r-l+1)
    return best
```

Binary search on answer (first True)

```
def first_true(lo, hi, chk):
    # find minimal x in [lo,hi] such that chk(x) is True; assumes monotonic
    while lo<hi:
        mid=(lo+hi)//2
        if chk(mid): hi=mid
        else: lo=mid+1
    return lo
```

Ternary search on unimodal real function

```
def ternary_search(l, r, f, iters=100):
    for _ in range(iters):
        m1 = l + (r-l)/3
        m2 = r - (r-l)/3
        if f(m1) < f(m2): l = m1
        else: r = m2
    return (l+r)/2
```

Count inversions via merge sort

```
def count_inversions(a):
    def sort(arr):
        if len(arr)<=1: return arr,0
        m=len(arr)//2
        L,cL=sort(arr[:m]); R,cR=sort(arr[m:])
        i=j=0; c=cL+cR; 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; c+=len(L)-i
        out+=L[i:]+R[j:]
        return out,c
    return sort(a)[1]
```

Meet-in-the-middle subset sum

```
from bisect import bisect_right
def can_subset_sum(a, target):
```

```
    mid=len(a)//2
    L=a[:mid]; R=a[mid:]
    sL=[0]
    for x in L:
        sL += [x+v for v in sL]
    sR=[0]
    for x in R:
        sR += [x+v for v in sR]
    sR.sort()
    for v in sL:
        if v>target: continue
        j=bisect_right(sR, target-v)-1
        if j>=0 and sR[j]<=target-v: return True
    return False
```

Bitset knapsack (sum achievable)

```
def knap_bitset(vals, S):
    bit=1  # bit i set ⇒ sum i achievable
    for x in vals:
        bit |= bit<<x
        bit &= (1<<(S+1))-1
    return [(bit>>i)&1 for i in range(S+1)]
```

Mo's algorithm (skeleton)

```
import math
def mos_algorithm(queries, a):
    # queries: list of (l,r,idx) inclusive; compute some function over range
    B=int(len(a)/max(1,math.sqrt(len(queries)))) or 1
    queries.sort(key=lambda q: (q[0]//B, q[1] if (q[0]//B)%2==0 else -q[1]))
    curL=0; curR=-1
    ans=[0]*len(queries)
    freq={}; cur=0
    def add(x):
        nonlocal cur
        c=freq.get(x,0); freq[x]=c+1
        # update cur with x added
    def remove(x):
        nonlocal cur
        c=freq[x]-=1
        # update cur with x removed
    for L,R,i in queries:
        while curL>L: curL-=1; add(a[curL])
        while curR<R: curR+=1; add(a[curR])
        while curL<L: remove(a[curL]); curL+=1
        while curR>R: remove(a[curR]); curR-=1
        ans[i]=cur
    return ans
```

Fenwick Tree (BIT)

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

Segment Tree (iterative, sum)

```
class SegTree:
    def __init__(self, a):
        n=1
        while n<len(a): n<<=1
        self.n=n; self.t=[0]*(2*n)
        self.t[n:n+len(a)]=a[:]
        for i in range(n-1,0,-1): self.t[i]=self.t[i<<1]+self.t[i<<1|1]
    def update(self, p, v):
```

```
        p+=self.n; self.t[p]=v
        p>>=1
        while p: self.t[p]=self.t[p<<1]+self.t[p<<1|1]; p>>=1
    def query(self, l, r):
        res=0; l+=self.n; r+=self.n+1
        while l<r:
            if l&1: res+=self.t[l]; l+=1
            if r&1: r-=1; res+=self.t[r]
            l>>=1; r>>=1
        return res
```

Sparse Table (RMQ min)

```
import math
def build_st(a):
    n=len(a); K=n.bit_length()
    st=[a[:]]; j=1
    while (1<<j)<=n:
        prev=st[-1]; span=1<<(j-1)
        st.append([min(prev[i], prev[i+span]) for i in range(n-(1<<j)+1)])
        j+=1
    lg=[0]*(n+1)
    for i in range(2,n+1): lg[i]=lg[i//2]+1
    return st,lg
def rmq(st,lg,l,r):
    j=lg[r-l+1]
    return min(st[j][l], st[j][r-(1<<j)+1])
```

DSU / Union-Find

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

Monotonic Queue (min in window)

```
from collections import deque
def sliding_min(a, k):
    dq=deque(); out=[]
    for i,x in enumerate(a):
        while dq and a[dq[-1]]>=x: dq.pop()
        dq.append(i)
        if dq[0]==i-k: dq.popleft()
        if i>=k-1: out.append(a[dq[0]])
    return out
```

2) Đồ thị cơ bản

BFS / DFS (iterative)

```
from collections import deque
def bfs(graph, s):
    n=len(graph); dist=[-1]*n; dist[s]=0
    q=deque([s])
    while q:
        u=q.popleft()
        for v in graph[u]:
            if dist[v]==-1:
                dist[v]=dist[u]+1; q.append(v)
    return dist

def dfs(graph, s):
    n=len(graph); seen=[False]*n; st=[s]; order=[]
    while st:
```

```
        u=st.pop()
        if seen[u]: continue
        seen[u]=True; order.append(u)
        for v in graph[u][::-1]:  # reverse for deterministic
            if not seen[v]: st.append(v)
    return order
```

Topological sort (Kahn)

```
from collections import deque
def topo_sort(n, edges):
    g=[[] for _ in range(n)]; indeg=[0]*n
    for u,v in edges: g[u].append(v); indeg[v]+=1
    q=deque([i for i in range(n) if indeg[i]==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)
    return order if len(order)==n else None
```

Dijkstra (min-heap)

```
import heapq
def dijkstra(n, adj, s):
    INF=10**18
    dist=[INF]*n; dist[s]=0
    pq=[(0,s)]
    while pq:
        d,u=heapq.heappop(pq)
        if d!=dist[u]: continue
        for v,w in adj[u]:
            nd=d+w
            if nd<dist[v]:
                dist[v]=nd; heapq.heappush(pq,(nd,v))
    return dist
```

0-1 BFS

```
from collections import deque
def zero_one_bfs(n, adj, s):
    INF=10**18; dist=[INF]*n; dist[s]=0
    dq=deque([s])
    while dq:
        u=dq.popleft()
        for v,w in adj[u]:  # w in {0,1}
            nd=dist[u]+w
            if nd<dist[v]:
                dist[v]=nd
                if w==0: dq.appendleft(v)
                else: dq.append(v)
    return dist
```

Bellman–Ford (detect negative cycle reachable)

```
def bellman_ford(n, edges, s):
    INF=10**18; dist=[INF]*n; dist[s]=0
    for _ in range(n-1):
        upd=False
        for u,v,w in edges:
            if dist[u]+w<dist[v]:
                dist[v]=dist[u]+w; upd=True
        if not upd: break
    neg=[False]*n
    for u,v,w in edges:
        if dist[u]+w<dist[v]:
            neg[v]=True
    return dist, neg
```

Floyd–Warshall (all-pairs shortest path)

```
def floyd_warshall(dist):
    # dist is n x n matrix; dist[i][i] should be 0
    n=len(dist)
    for k in range(n):
        dk=dist[k]
```

```
        for i in range(n):
            dik=dist[i][k]
            if dik==float('inf'): continue
            for j in range(n):
                v=dik+dk[j]
                if v<dist[i][j]: dist[i][j]=v
    return dist
```

Kruskal MST

```
def kruskal(n, edges):
    # edges: (w,u,v)
    edges.sort()
    dsu=DSU(n); total=0; used=[]
    for w,u,v in edges:
        if dsu.union(u,v):
            total+=w; used.append((u,v,w))
    return total, used
```

Tarjan bridges & articulation points

```
def bridges_aps(n, g):
    timer=0
    tin=[-1]*n; low=[0]*n; is_art=[False]*n
    bridges=[]
    st=[(0,-1,0)]  # (u, parent, state) state 0 enter, 1 exit
    parent=[-1]*n; it=[0]*n; child=[0]*n
    # convert to iterative DFS
    for r in range(n):
        if tin[r]!=-1: continue
        stack=[(r,-1,0)]
        while stack:
            u,p,state=stack.pop()
            if state==0:
                parent[u]=p; tin[u]=low[u]=timer; timer+=1
                stack.append((u,p,1))
                for v in g[u]:
                    if v==p: continue
                    if tin[v]==-1:
                        child[u]+=1
                        stack.append((v,u,0))
                    else:
                        low[u]=min(low[u], tin[v])
            else:
                for v in g[u]:
                    if v==parent[u]: continue
                    if parent[v]==u:
                        low[u]=min(low[u], low[v])
                        if low[v]>tin[u]: bridges.append((u,v))
                if parent[u]!=-1 and low[u]>=tin[parent[u]]: is_art[parent[u]]=True
        if child[r]>1: is_art[r]=True
    return bridges, [i for i,x in enumerate(is_art) if x]
```

SCC (Kosaraju)

```
def scc_kosaraju(n, edges):
    g=[[] for _ in range(n)]; rg=[[] for _ in range(n)]
    for u,v in edges: g[u].append(v); rg[v].append(u)
    seen=[False]*n; order=[]
    def dfs(u):
        seen[u]=True
        for v in g[u]:
            if not seen[v]: dfs(v)
        order.append(u)
    for i in range(n):
        if not seen[i]: dfs(i)
    comp=[-1]*n; k=0
    def rdfs(u,c):
        comp[u]=c
        for v in rg[u]:
            if comp[v]==-1: rdfs(v,c)
    for u in reversed(order):
        if comp[u]==-1:
            rdfs(u,k); k+=1
    return comp, k
```

DP on DAG (đếm số đường từ s đến t)

```
def count_paths_dag(n, edges, s, t):
    order=topo_sort(n, edges)
    if not order: return 0
    g=[[] for _ in range(n)]
    for u,v in edges: g[u].append(v)
    dp=[0]*n; dp[s]=1
    pos=[0]*n
    for i,u in enumerate(order): pos[u]=i
    for u in order:
        for v in g[u]:
            dp[v]+=dp[u]
    return dp[t]
```

Euler trail (Hierholzer, undirected)

```
from collections import defaultdict
def euler_trail_undirected(edges):
    # edges: list of (u,v)
    g=defaultdict(list)
    for i,(u,v) in enumerate(edges):
        g[u].append((v,i)); g[v].append((u,i))
    start=edges[0][0]
    st=[start]; used=[False]*len(edges); path=[]
    while st:
        u=st[-1]
        while g[u] and used[g[u][-1][1]]: g[u].pop()
        if not g[u]:
            path.append(u); st.pop()
        else:
            v,i=g[u].pop()
            if not used[i]:
                used[i]=True; st.append(v)
    return path[::-1]
```

3) Luồng & ghép cặp

Dinic (Max Flow)

```
from collections import deque
class Dinic:
    def __init__(self, n):
        self.n=n; self.to=[]; self.cap=[]; self.next=[]; self.head=[-1]*n
    def add_edge(self,u,v,c):
        self.to.append(v); self.cap.append(c); self.next.append(self.head[u]); self.head[u]=len(self.to)-1
        self.to.append(u); self.cap.append(0); self.next.append(self.head[v]); self.head[v]=len(self.to)-1
    def maxflow(self,s,t):
        flow=0
        while True:
            level=[-1]*self.n; q=deque([s]); level[s]=0
            while q:
                u=q.popleft()
                i=self.head[u]
                while i!=-1:
                    v=self.to[i]
                    if self.cap[i]>0 and level[v]<0:
                        level[v]=level[u]+1; q.append(v)
                    i=self.next[i]
            if level[t]<0: break
            it=self.head[:]
            def dfs(u,f):
                if u==t: return f
                i=it[u]
                while i!=-1:
                    it[u]=i
                    v=self.to[i]
                    if self.cap[i]>0 and level[u]+1==level[v]:
                        ret=dfs(v, min(f, self.cap[i]))
                        if ret:
                            self.cap[i]-=ret; self.cap[i^1]+=ret; return ret
                    i=self.next[i]; it[u]=i
                return 0
            while True:
                pushed=dfs(s, 10**18)
```

```
                if not pushed: break
                flow+=pushed
        return flow
```

Min-Cost Max-Flow (SPFA-based)

```
from collections import deque
def min_cost_max_flow(n, edges, s, t):
    # edges: (u,v,cap,cost)
    g=[[] for _ in range(n)]
    def add(u,v,c,w):
        g[u].append([v,c,w,len(g[v])])
        g[v].append([u,0,-w,len(g[u])-1])
    for u,v,c,w in edges: add(u,v,c,w)
    flow=cost=0
    INF=10**18
    while True:
        dist=[INF]*n; inq=[False]*n; pv=[(-1,-1)]*n
        dist[s]=0; dq=deque([s]); inq[s]=True
        while dq:
            u=dq.popleft(); inq[u]=False
            for i,(v,c,w,rev) in enumerate(g[u]):
                if c and dist[u]+w<dist[v]:
                    dist[v]=dist[u]+w; pv[v]=(u,i)
                    if not inq[v]:
                        dq.append(v); inq[v]=True
        if dist[t]==INF: break
        addf=INF; v=t
        while v!=s:
            u,i=pv[v]; addf=min(addf, g[u][i][1]); v=u
        v=t
        while v!=s:
            u,i=pv[v]; e=g[u][i]; e[1]-=addf; g[v][e[3]][1]+=addf; v=u
        flow+=addf; cost+=addf*dist[t]
    return flow, cost
```

Hopcroft–Karp (matching bipartite)

```
from collections import deque
def hopcroft_karp(U, V, adj):
    INF=10**18
    pairU=[-1]*U; pairV=[-1]*V; dist=[0]*U
    def bfs():
        q=deque()
        for u in range(U):
            if pairU[u]==-1:
                dist[u]=0; q.append(u)
            else: dist[u]=INF
        D=INF
        while q:
            u=q.popleft()
            if dist[u]<D:
                for v in adj[u]:
                    if pairV[v]==-1: D=dist[u]+1
                    elif dist[pairV[v]]==INF:
                        dist[pairV[v]]=dist[u]+1; q.append(pairV[v])
        return D!=INF
    def dfs(u):
        for v in adj[u]:
            if pairV[v]==-1 or (dist[pairV[v]]==dist[u]+1 and dfs(pairV[v])):
                pairU[u]=v; pairV[v]=u; return True
        dist[u]=10**18; return False
    matching=0
    while bfs():
        for u in range(U):
            if pairU[u]==-1 and dfs(u): matching+=1
    return matching, pairU, pairV
```

Hungarian (assignment O(n^3))

```
def hungarian(a):
    # a: cost matrix n x n (minimize)
    n=len(a)
    u=[0]*(n+1); v=[0]*(n+1); p=[0]*(n+1); way=[0]*(n+1)
    for i in range(1,n+1):
        p[0]=i; j0=0
        minv=[float('inf')]*(n+1); used=[False]*(n+1)
```

```
        while True:
            used[j0]=True; i0=p[j0]; delta=float('inf'); j1=0
            for j in range(1,n+1):
                if used[j]: continue
                cur=a[i0-1][j-1]-u[i0]-v[j]
                if cur<minv[j]: minv[j]=cur; way[j]=j0
                if minv[j]<delta: delta=minv[j]; j1=j
            for j in range(n+1):
                if used[j]: u[p[j]]+=delta; v[j]-=delta
                else: minv[j]-=delta
            j0=j1
            if p[j0]==0: break
        while True:
            j1=way[j0]; p[j0]=p[j1]; j0=j1
            if j0==0: break
    match=[0]*n
    for j in range(1,n+1):
        if p[j]>0: match[p[j]-1]=j-1
    return match
```

4) Cây & kỹ thuật trên cây

LCA – Binary Lifting

```
import math
def build_lca(n, root, g):
    LOG=(n).bit_length()
    up=[[ -1]*n for _ in range(LOG)]
    depth=[0]*n
    st=[(root,-1,0)]
    order=[]
    while st:
        u,p,typ=st.pop()
        if typ==0:
            order.append(u)
            if p!=-1: depth[u]=depth[p]+1
            up[0][u]=p
            st.append((u,p,1))
            for v in g[u]:
                if v==p: continue
                st.append((v,u,0))
    for k in range(1,LOG):
        for v in range(n):
            up[k][v] = up[k-1][up[k-1][v]] if up[k-1][v]!=-1 else -1
    return up, depth

def lca(u,v,up,depth):
    if depth[u]<depth[v]: u,v=v,u
    LOG=len(up)
    diff=depth[u]-depth[v]
    for k in range(LOG):
        if diff>>k &1: u=up[k][u]
    if u==v: return u
    for k in range(LOG-1,-1,-1):
        if up[k][u]!=up[k][v]:
            u=up[k][u]; v=up[k][v]
    return up[0][u]
```

Centroid Decomposition (skeleton)

```
def centroid_decomp(g):
    n=len(g); sz=[0]*n; dead=[False]*n; parent=[-1]*n
    def getsz(u,p):
        sz[u]=1
        for v in g[u]:
            if v==p or dead[v]: continue
            getsz(v,u); sz[u]+=sz[v]
    def getcent(u,p,tot):
        for v in g[u]:
            if v==p or dead[v]: continue
            if sz[v]*2>tot: return getcent(v,u,tot)
        return u
    def build(u,p):
        getsz(u,-1); c=getcent(u,-1,sz[u])
        dead[c]=True; parent[c]=p
```

```
        for v in g[c]:
            if not dead[v]: build(v,c)
        return c
    root=build(0,-1)
    return parent, root
```

Rerooting DP (sum of distances)

```
def rerooting_sum_dist(n, g):
    sz=[1]*n; dp=[0]*n
    order=[]; st=[(0,-1,0)]
    while st:
        u,p,t=st.pop()
        if t==0:
            st.append((u,p,1))
            for v in g[u]:
                if v==p: continue
                st.append((v,u,0))
        else:
            for v in g[u]:
                if v==p: continue
                sz[u]+=sz[v]; dp[u]+=dp[v]+sz[v]
    ans=[0]*n
    ans[0]=dp[0]
    st=[(0,-1)]
    while st:
        u,p=st.pop()
        for v in g[u]:
            if v==p: continue
            ans[v]=ans[u] - sz[v] + (n - sz[v])
            st.append((v,u))
    return ans
```

5) Quy hoạch động & tối ưu hoá

0/1 Knapsack (O(nS))

```
def knapsack_01(items, S):
    dp=[0]*(S+1)
    for w,val in items:
        for s in range(S,w-1,-1):
            dp[s]=max(dp[s], dp[s-w]+val)
    return max(dp)
```

Digit DP (count numbers ≤ N without digit 4)

```
from functools import lru_cache
def count_no4(N):
    s=str(N)
    @lru_cache(None)
    def dp(i, tight, started):
        if i==len(s): return 1 if started else 1
        lim = int(s[i]) if tight else 9
        res=0
        for d in range(lim+1):
            if d==4: continue
            res += dp(i+1, tight and d==lim, started or d!=0)
        return res
    return dp(0, True, False)
```

Convex Hull Trick (monotonic slopes, query increasing x)

```
class CHT:
    # Lines: y = m x + b; insert m increasing; query x increasing.
    def __init__(self):
        self.m=[]; self.b=[]; self.p=[]  # intersection x with prev
    def is_bad(self, m1,b1, m2,b2, m3,b3):
        return (b3-b1)*(m1-m2) <= (b2-b1)*(m1-m3)
    def add(self, m, b):
        while len(self.m)>=2 and self.is_bad(self.m[-2],self.b[-2], self.m[-1],self.b[-1], m,b):
            self.m.pop(); self.b.pop(); self.p.pop()
        if self.m:
            x=(self.b[-1]-b)/(m-self.m[-1])
            self.p.append(x)
        else:
            self.p.append(float('-inf'))
```

```
        self.m.append(m); self.b.append(b)
    def query(self, x):
        import bisect
        i = len(self.p)-1 if x>=self.p[-1] else bisect.bisect_right(self.p, x)-1
        return self.m[i]*x + self.b[i]
```

Divide & Conquer DP Optimization (skeleton)

```
# Compute dp[i][j] with opt monotonicity: opt[i][j] ≤ opt[i][j+1]
def dc_optimize(cost, n, k):
    dp_prev=[0]*n; dp_cur=[0]*n
    def compute(l,r, optl, optr):
        if l>r: return
        m=(l+r)//2; best=(-1,10**18)
        for kopt in range(optl, min(m,optr)+1):
            val = dp_prev[kopt] + cost(kopt,m)
            if val<best[1]: best=(kopt,val)
        dp_cur[m]=best[1]; opt=best[0]
        compute(l,m-1,optl,opt); compute(m+1,r,opt,optr)
    for _ in range(k):
        compute(0,n-1,0,n-1)
        dp_prev,dp_cur=dp_cur, [0]*n
    return dp_prev
```

SOS DP (sum over supersets)

```
def sos_dp(f):
    # f over bitmasks size 2^n; returns F[mask]=sum f[s] for s ⊆ mask
    F=f[:]; n=len(F).bit_length()-1
    for i in range(n):
        for mask in range(1<<n):
            if mask>>i &1: F[mask]+=F[mask^(1<<i)]
    return F
```

6) Xâu & chuỗi ký tự

KMP

```
def kmp_table(p):
    n=len(p); lps=[0]*n; j=0
    for i in range(1,n):
        while j 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_table(p); j=0; pos=[]
    for i,ch in enumerate(s):
        while j and ch!=p[j]: j=lps[j-1]
        if ch==p[j]: j+=1
        if j==len(p): pos.append(i-j+1); j=lps[j-1]
    return pos
```

Z-Algorithm

```
def z_algorithm(s):
    n=len(s); z=[0]*n; l=r=0
    for i in range(1,n):
        if i<=r: z[i]=min(r-i+1,z[i-l])
        while i+z[i]<n and s[z[i]]==s[i+z[i]]: z[i]+=1
        if i+z[i]-1>r: l=i; r=i+z[i]-1
    z[0]=n; return z
```

Rolling hash (double)

```
def build_hash(s, B1=911382323, M1=10**9+7, B2=972663749, M2=10**9+9):
    n=len(s)
    p1=[1]*(n+1); h1=[0]*(n+1)
    p2=[1]*(n+1); h2=[0]*(n+1)
    for i,ch in enumerate(map(ord,s),1):
        p1[i]=(p1[i-1]*B1)%M1; h1[i]=(h1[i-1]*B1+ch)%M1
        p2[i]=(p2[i-1]*B2)%M2; h2[i]=(h2[i-1]*B2+ch)%M2
    def get(l,r):
        x1=(h1[r]-h1[l]*p1[r-l])%M1
        x2=(h2[r]-h2[l]*p2[r-l])%M2
        return (x1,x2)
    return get
```

Aho–Corasick (multi-pattern)

```
from collections import deque
def aho_build(words):
    nxt=[{ }]; link=[0]; out=[[]]
    for idx,w in enumerate(words):
        v=0
        for ch in w:
            v = nxt[v].setdefault(ch, len(nxt));
            if v==len(nxt): nxt.append({}); link.append(0); out.append([])
        out[v].append(idx)
    q=deque()
    for ch,v in nxt[0].items():
        q.append(v); link[v]=0
    while q:
        v=q.popleft()
        for ch,u in list(nxt[v].items()):
            q.append(u)
            j=link[v]
            while j and ch not in nxt[j]: j=link[j]
            link[u]=nxt[j].get(ch,0)
            out[u]+=out[link[u]]
    return nxt,link,out

def aho_search(s, aut):
    nxt,link,out=aut; v=0; res=[]
    for i,ch in enumerate(s):
        while v and ch not in nxt[v]: v=link[v]
        v=nxt[v].get(ch,0)
        for id in out[v]: res.append((i,id))
    return res
```

Suffix Array (doubling) + LCP (Kasai)

```
def suffix_array(s):
    n=len(s); k=1
    sa=list(range(n)); r=[ord(c) for c in s]; tmp=[0]*n
    while True:
        sa.sort(key=lambda i:(r[i], r[i+k] if i+k<n else -1))
        tmp[sa[0]]=0
        for i in range(1,n):
            prev=sa[i-1]; cur=sa[i]
            tmp[cur]=tmp[prev]+((r[prev], r[prev+k] if prev+k<n else -1)!=(r[cur], r[cur+k] if cur+k<n else
        r,tmp=tmp,r
        if r[sa[-1]]==n-1: break
        k<<=1
    return sa

def lcp_kasai(s, sa):
    n=len(s); rank=[0]*n
    for i,p in enumerate(sa): rank[p]=i
    lcp=[0]*(n-1); h=0
    for i in range(n):
        j=sa[rank[i]-1]
        while i+h<n and j+h<n and s[i+h]==s[j+h]: h+=1
        if rank[i]>0: lcp[rank[i]-1]=h
        if h: h-=1
    return lcp
```

Suffix Automaton (SAM) – minimal

```
def sam_build(s):
    link=[-1]; next=[{}]; length=[0]; last=0
    def add(ch):
        nonlocal last
        cur=len(length); length.append(length[last]+1); link.append(0); next.append({})
        p=last
        while p!=-1 and ch not in next[p]:
            next[p][ch]=cur; p=link[p]
        if p==-1: link[cur]=0
        else:
            q=next[p][ch]
            if length[p]+1==length[q]: link[cur]=q
            else:
                clone=len(length); length.append(length[p]+1); link.append(link[q]); next.append(next[q].cop
                while p!=-1 and next[p].get(ch, -1)==q:
```

```
                    next[p][ch]=clone; p=link[p]
                link[q]=link[cur]=clone
        last=cur
    for ch in s: add(ch)
    return link, next, length
```

Manacher (palindrome radii)

```
def manacher(s):
    T='|'+'|'.join(s)+'|'
    n=len(T); P=[0]*n; c=r=0
    for i in range(n):
        mir=2*c-i
        if i<r: P[i]=min(r-i, P[mir])
        while 0<=i-P[i]-1 and i+P[i]+1<n and T[i-P[i]-1]==T[i+P[i]+1]:
            P[i]+=1
        if i+P[i]>r: c=i; r=i+P[i]
    # radius in original string at centers
    return P
```

Booth (minimal rotation)

```
def minimal_rotation(s):
    s+=s; n=len(s); i=0; ans=0
    while i<n//2:
        ans=i; j=i+1; k=0
        while j<n and s[i+k]==s[j+k]:
            k+=1
            if k==n//2: return ans
        if s[i+k]>s[j+k]: i=j
        else: i=i+k+1
    return ans
```

7) Hình học tính toán

CCW & Segment orientation

```
def cross(a,b): return a[0]*b[1]-a[1]*b[0]
def sub(a,b): return (a[0]-b[0], a[1]-b[1])
def ccw(a,b,c): return cross(sub(b,a), sub(c,a))  # >0 left, <0 right, 0 collinear

def on_segment(a,b,p):
    return min(a[0],b[0])<=p[0]<=max(a[0],b[0]) and min(a[1],b[1])<=p[1]<=max(a[1],b[1])

def seg_intersect(a,b,c,d):
    o1=ccw(a,b,c); o2=ccw(a,b,d); o3=ccw(c,d,a); o4=ccw(c,d,b)
    if o1==0 and on_segment(a,b,c): return True
    if o2==0 and on_segment(a,b,d): return True
    if o3==0 and on_segment(c,d,a): return True
    if o4==0 and on_segment(c,d,b): return True
    return (o1>0)^(o2>0) and (o3>0)^(o4>0)
```

Convex Hull (Monotone Chain)

```
def convex_hull(pts):
    pts=sorted(set(pts))
    if len(pts)<=1: return pts
    def half(seq):
        H=[]
        for p in seq:
            while len(H)>=2 and ccw(H[-2], H[-1], p)<=0: H.pop()
            H.append(p)
        return H
    L=half(pts)
    U=half(reversed(pts))
    return L[:-1]+U[:-1]
```

Rotating calipers (diameter of convex polygon)

```
def polygon_diameter(poly):
    # poly must be convex, CCW
    def dist2(a,b): return (a[0]-b[0])**2 + (a[1]-b[1])**2
    n=len(poly); if n<2: return 0
    j=1; best=0
    for i in range(n):
        ni=(i+1)%n
```

```
        while abs(cross(sub(poly[ni], poly[i]), sub(poly[(j+1)%n], poly[j])))> \
              abs(cross(sub(poly[ni], poly[i]), sub(poly[j], poly[j]))):
            j=(j+1)%n
        best=max(best, dist2(poly[i], poly[j]))
    return best**0.5
```

Point in polygon (ray casting)

```
def point_in_polygon(poly, p):
    x,y=p; inside=False; n=len(poly)
    for i in range(n):
        x1,y1=poly[i]; x2,y2=poly[(i+1)%n]
        if (y1>y)!=(y2>y):
            xint = (x2-x1)*(y-y1)/(y2-y1+1e-18)+x1
            if x<xint: inside=not inside
    return inside
```

8) Số học & lý thuyết số

Sieve & SPF

```
def sieve_spf(n):
    spf=list(range(n+1))
    for i in range(2,int(n**0.5)+1):
        if spf[i]==i:
            step=i
            for j in range(i*i, n+1, i):
                if spf[j]==j: spf[j]=i
    primes=[i for i in range(2,n+1) if spf[i]==i]
    return primes, spf
```

EGCD, modinv, powmod

```
def egcd(a,b):
    if b==0: return (1,0,a)
    x,y,g=egcd(b, a%b)
    return (y, x-(a//b)*y, g)
def modinv(a,m):
    x,y,g=egcd(a,m)
    if g!=1: return None
    return x%m
def powmod(a,e,m):
    res=1%m
    while e:
        if e&1: res=res*a%m
        a=a*a%m; e>>=1
    return res
```

Chinese Remainder Theorem (pairwise)

```
def crt_pair(a1,m1,a2,m2):
    # solve x ≡ a1 (mod m1), x ≡ a2 (mod m2)
    x1,y1,g=egcd(m1,m2)
    if (a2-a1)%g!=0: return None,None
    l=m1//g*m2
    x=(a1 + (a2-a1)//g * x1 % (m2//g) * m1) % l
    return x,l
```

Euler's Totient sieve

```
def phi_sieve(n):
    phi=list(range(n+1))
    for i in range(2,n+1):
        if phi[i]==i:
            for j in range(i, n+1, i):
                phi[j]-=phi[j]//i
    return phi
```

Miller–Rabin (deterministic for 64-bit)

```
def is_probable_prime(n):
    if n<2: return False
    small = [2,3,5,7,11,13,17,19,23,29,31,37]
    for p in small:
        if n%p==0: return n==p
    d=n-1; s=0
```

```
    while d%2==0: d//=2; s+=1
    def check(a):
        x=pow(a,d,n)
        if x==1 or x==n-1: return True
        for _ in range(s-1):
            x=(x*x)%n
            if x==n-1: return True
        return False
    for a in [2,325,9375,28178,450775,9780504,1795265022]:
        if a%(n)==0: return True
        if not check(a): return False
    return True
```

Pollard's Rho (factorization)

```
import random, math
def pollards_rho(n):
    if n%2==0: return 2
    if is_probable_prime(n): return n
    while True:
        c=random.randrange(1,n)
        f=lambda x:(x*x+c)%n
        x=y=random.randrange(0,n); d=1
        while d==1:
            x=f(x); y=f(f(y))
            d=math.gcd(abs(x-y), n)
        if d!=n: return d

def factor(n):
    if n==1: return []
    if is_probable_prime(n): return [n]
    d=pollards_rho(n)
    return factor(d)+factor(n//d)
```

Linear Diophantine: ax + by = c

```
def solve_diophantine(a,b,c):
    x0,y0,g=egcd(abs(a),abs(b))
    if c%g: return None
    x0*=c//g; y0*=c//g
    if a<0: x0=-x0
    if b<0: y0=-y0
    return x0,y0,g  # general: x=x0 + k*(b/g), y=y0 - k*(a/g)
```

nCk mod prime with factorial precompute

```
MOD=10**9+7
def precompute_fact(n, MOD=MOD):
    fac=[1]*(n+1); ifac=[1]*(n+1)
    for i in range(1,n+1): fac[i]=fac[i-1]*i%MOD
    ifac[n]=pow(fac[n], MOD-2, MOD)
    for i in range(n,0,-1): ifac[i-1]=ifac[i]*i%MOD
    return fac, ifac
def nCk(n,k,fac,ifac,MOD=MOD):
    if k<0 or k>n: return 0
    return fac[n]*ifac[k]%MOD*ifac[n-k]%MOD
```

Kitamasa (linear recurrence)

```
def poly_mul(a,b,mod):
    n=len(a); res=[0]*(2*n-1)
    for i,x in enumerate(a):
        for j,y in enumerate(b):
            res[i+j]=(res[i+j]+x*y)%mod
    return res
def poly_mod(p, c, mod):
    # reduce degree using recurrence coefficients c (x^n = c[0] x^{n-1}+...)
    n=len(c); res=p[:]
    for i in range(len(res)-1, n-1, -1):
        coef=res[i]
        if coef:
            for j in range(n):
                res[i-1-j]=(res[i-1-j]+coef*c[j])%mod
    return res[:n]
def kitamasa(k, c, a0, n, mod):
    # a[n] from order-k recurrence: a[i]=sum c[j]*a[i-1-j]
    if n<k: return a0[n]%mod
```

```
    pol=[0]*k; pol[0]=1  # x^0
    base=[0]*k; base[1]=1  # x
    c=[x%mod for x in c]
    def mul(x,y): return poly_mod(poly_mul(x,y,mod), c, mod)
    def pw(p,e):
        res=[0]*k; res[0]=1
        while e:
            if e&1: res=mul(res,p)
            p=mul(p,p); e>>=1
        return res
    coef=pw(base, n)
    ans=0
    for i in range(k):
        ans=(ans + coef[i]*a0[i])%mod
    return ans
```

9) Kỹ thuật khác

2-SAT (implication graph + SCC)

```
def two_sat(n, clauses):
    # variables x in [0..n-1], literals as (var, is_true)
    m=2*n
    g=[[] for _ in range(m)]; rg=[[] for _ in range(m)]
    def var(u, val): return u*2 + (0 if val else 1)  # x -> 0, ~x ->1
    for (a,va),(b,vb) in clauses:
        A=var(a,va); NA=A^1
        B=var(b,vb); NB=B^1
        g[NA].append(B); g[NB].append(A)
        rg[B].append(NA); rg[A].append(NB)
    # Kosaraju
    seen=[False]*m; order=[]
    def dfs(u):
        seen[u]=True
        for v in g[u]:
            if not seen[v]: dfs(v)
        order.append(u)
    for i in range(m):
        if not seen[i]: dfs(i)
    comp=[-1]*m; k=0
    def rdfs(u,c):
        comp[u]=c
        for v in rg[u]:
            if comp[v]==-1: rdfs(v,c)
    for u in reversed(order):
        if comp[u]==-1: rdfs(u,k); k+=1
    assign=[False]*n
    for i in range(n):
        if comp[2*i]==comp[2*i+1]: return None
        assign[i]= comp[2*i]>comp[2*i+1]
    return assign
```

Grundy (Sprague–Grundy) skeleton

```
from functools import lru_cache
def mex(s):
    i=0
    while i in s: i+=1
    return i
def grundy(game_moves, state):
    # game_moves(state) -> iterable of next states
    @lru_cache(None)
    def g(s):
        return mex({g(ns) for ns in game_moves(s)})
    return g(state)
```

Floyd/Brent cycle detection for f(x)

```
def floyd_cycle(f, x0):
    tort=f(x0); hare=f(f(x0))
    while tort!=hare:
        tort=f(tort); hare=f(f(hare))
    mu=0; hare=x0
    while tort!=hare:
        tort=f(tort); hare=f(hare); mu+=1
    lam=1; hare=f(tort)
```

```
    while tort!=hare:
        hare=f(hare); lam+=1
    return mu, lam

def brent_cycle(f, x0):
    power=lam=1; tort=x0; hare=f(x0)
    while tort!=hare:
        if power==lam: tort=hare; power*=2; lam=0
        hare=f(hare); lam+=1
    mu=0; tort=hare=x0
    for _ in range(lam):
        hare=f(hare)
    while tort!=hare:
        tort=f(tort); hare=f(hare); mu+=1
    return mu, lam
```

Quickselect (k-th smallest, 0-indexed)

```
import random
def quickselect(a, k):
    l, r = 0, len(a)-1
    while True:
        if l==r: return a[l]
        piv=a[random.randint(l,r)]
        i,j, t=l,r,l
        while t<=j:
            if a[t]<piv: a[i],a[t]=a[t],a[i]; i+=1; t+=1
            elif a[t]>piv: a[t],a[j]=a[j],a[t]; j-=1
            else: t+=1
        if k<i: r=i-1
        elif k>j: l=j+1
        else: return a[k]
```

Coordinate compression

```
def compress(arr):
    vals=sorted(set(arr))
    idx={v:i for i,v in enumerate(vals)}
    return [idx[x] for x in arr], vals
```

Ghi chú: Một số thuật toán nâng cao (HLD chi tiết, DSU rollback, Eertree, Half-plane, Minkowski...) cần nhiều mã hơn – có thể bổ sung theo nhu cầu bài.