UTCC

มหาวิทยาลัยหอการค้าไทย

The University of the Thai Chamber of Commerce

Team Name: Harbourspace_199

Participants:

Giorgi Kurtanidze
Levan Dalbashvili
Irakli Kereleishvili

Binary Search

Use when: The problem requires efficient search in a sorted/monotonic structure (e.g., array or function where properties flip from False to True), or optimizing a value (min/max) via a feasibility check that holds monotonically across a range. **Spot it if:** Data is sorted/pre-sortable for lookups, or you can binary-search answers by testing "is X possible?" and adjusting bounds—avoids brute force for large N (e.g., 10^9). **Typical for:** Element positions/counts in arrays, min/max optimization like "smallest K where condition holds," resource allocation, or scheduling with check functions.

Using bisect (Built-in)

Use when: Working with sorted arrays and need fast lookups. **Time:** $O(\log N)$

bisect_left: Finds leftmost (first) position where element $\geq$ target.

```
from bisect import bisect_left
arr = [1, 3, 3, 5, 7]
print(bisect_left(arr, 3))  # 1 (first element >= 3)
print(bisect_left(arr, 4))  # 3 (first element >= 4)
print(bisect_left(arr, 0))  # 0 (before first element)
print(bisect_left(arr, 10)) # 5 (after all elements)
```

bisect_right: Finds rightmost position where element $>$ target (first position after all equal elements).

```
from bisect import bisect_right
arr = [1, 3, 3, 5, 7]
print(bisect_right(arr, 3))  # 3 (first element > 3)
print(bisect_right(arr, 4))  # 3 (first element > 4)
print(bisect_right(arr, 7))  # 5 (after last element)
```

Binary Search on Answer

Use when: You need to find min/max value satisfying a condition (check function). Pattern: condition changes from False to True (or True to False) as value increases. **Time:** $O(\log N \cdot T)$ where T is check function complexity.

Find Maximum x where check(x) is True:

```
def binary_search_max(left, right):a
    # left = minimum possible value
    # right = maximum possible value + 1
    while left < right:
        mid = (left + right) // 2
        if check(mid):
            left = mid + 1  # mid works, try higher
        else:
            right = mid     # mid doesn't work, go lower
    return left - 1         # last True
```

Find Minimum x where check(x) is True:

```
def binary_search_min(left, right):
    # left = minimum possible value
    # right = maximum possible value + 1
    while left < right:
        mid = (left + right) // 2
        if check(mid):
            right = mid     # mid works, try lower
        else:
            left = mid + 1  # mid doesn't work, go higher
    return left             # first True
```

Example: Find max/min x where $x^2 \leq 50$

```
def check(x):
    return x*x <= 50

# Max x where x^2 <= 50
mx = binary_search_max(0, 101)
print("Max x:", mx)  # 7 (because 7*7=49, 8*8=64>50)
```

```
# Min x where x^2 <= 50
mn = binary_search_min(0, 101)
print("Min x:", mn)  # 0 (because 0*0=0 <= 50)
```

Scratch Implementation

Use when: Custom binary search without bisect library. Common pattern for contest problems with custom conditions.

```
l = 0
r = len(arr)
while r > l + 1:
    mid = (l + r) // 2
    if is_possible(mid):  # your check function
        l = mid
    else:
        r = mid
# Answer is at index l
```

Connected Components

Use when: You need to count separate parts of a graph or check if nodes are connected. Typical for: islands, networks, group counting, cluster problems.

DFS / BFS – Undirected Graph Time: $O(N + M)$

DFS Implementation:

```
def count_cc_dfs(n, edges):
    g = [[] for _ in range(n+1)]
    for u,v in edges:
        g[u].append(v); g[v].append(u)
    vis = [0]*(n+1)
    def dfs(u):
        st=[u]
        while st:
            x=st.pop()
            if vis[x]: continue
            vis[x]=1
            for v in g[x]:
                if not vis[v]: st.append(v)
    c=0
    for i in range(1,n+1):
        if not vis[i]: dfs(i); c+=1
    return c
```

BFS Implementation:

```
def count_cc_bfs(n, edges):
    from collections import deque
    g = [[] for _ in range(n+1)]
    for u,v in edges:
        g[u].append(v); g[v].append(u)
    vis = [0]*(n+1)
    def bfs(start):
        q = deque([start])
        vis[start] = 1
        while q:
            u = q.popleft()
            for v in g[u]:
                if not vis[v]:
                    vis[v] = 1
                    q.append(v)
    c = 0
    for i in range(1, n+1):
        if not vis[i]:
            bfs(i)
            c += 1
    return c
```

Shortest Path Algorithms

Use when: Finding minimal distance or cost between nodes. Pick depending on edge type (unweighted, weighted, or negative weights).

Unweighted (BFS)

Use when: All edges have equal cost (1). **Time:** $O(N + M)$

```
def shortest_path_bfs(n, edges, s, t):
    g=[[] for _ in range(n+1)]
    for u,v in edges:
        g[u].append(v); g[v].append(u)
    d=[1e9]*(n+1); p=[-1]*(n+1)
    q=[s]; d[s]=0
    for u in q:
        for v in g[u]:
            if d[v]>1e8:
                d[v]=d[u]+1; p[v]=u; q.append(v)
    if d[t]==1e9: return [], 1e9
    path=[]; x=t
    while x!=-1: path.append(x); x=p[x]
    return path[::-1], d[t]
```

Dijkstra

Use when: All weights are non-negative. **Time:** $O((N + M) \log N)$

```
def dijkstra(n, edges, s):
    import heapq
    g=[[] for _ in range(n+1)]
    for u,v,w in edges:
        g[u].append((v,w)); g[v].append((u,w))
    d=[1e18]*(n+1); p=[-1]*(n+1)
    d[s]=0; pq=[(0,s)]
    while pq:
        du,u=heapq.heappop(pq)
        if du!=d[u]: continue
        for v,w in g[u]:
            if d[v]>d[u]+w:
                d[v]=d[u]+w; p[v]=u
                heapq.heappush(pq,(d[v],v))
    return d,p
```

Bellman-Ford

Use when: Negative weights may appear, or you must detect negative cycles. **Time:** $O(NM)$

```
def bellman(n, edges, s):
    d=[1e18]*(n+1); d[s]=0
    for _ in range(n-1):
        for u,v,w in edges:
            if d[u]+w<d[v]: d[v]=d[u]+w
    neg=False
    for u,v,w in edges:
        if d[u]+w<d[v]: neg=True
    return d, neg
```

Connectivity Checks

Use when: You must verify if the graph is one connected piece or find weak points (bridges).

Is Connected

Checks if the graph is fully connected. **Time:** $O(N + M)$

```
def is_connected(n, edges):
    from collections import deque
    g=[[] for _ in range(n+1)]
    for u,v in edges:
        g[u].append(v); g[v].append(u)
    vis=[0]*(n+1)
    q=deque([1]); vis[1]=1; c=1
    while q:
        u=q.popleft()
        for v in g[u]:
            if not vis[v]:
                vis[v]=1; q.append(v); c+=1
    return c==n
```

Find Bridges (Tarjan)

Use when: You need edges that, if removed, increase the number of connected components (critical connections). **Time:** $O(N + M)$

```
def bridges(n, edges):
    g=[[] for _ in range(n+1)]
    for u,v in edges:
        g[u].append(v); g[v].append(u)
    tin=[0]*(n+1); low=[0]*(n+1)
    vis=[0]*(n+1); t=0; br=[]
    def dfs(u,p):
        nonlocal t
        vis[u]=1; t+=1
        tin[u]=low[u]=t
        for v in g[u]:
            if v==p: continue
            if vis[v]: low[u]=min(low[u],tin[v])
            else:
                dfs(v,u)
                low[u]=min(low[u],low[v])
                if low[v]>tin[u]: br.append((u,v))
    for i in range(1,n+1):
        if not vis[i]: dfs(i,-1)
    return br
```

Pathfinding and Existence

Use when: You need to know if any path exists or to list all possible paths.

All Paths (DFS)

Useful for small graphs (exponential complexity).

```
def all_paths(n, edges, s, t):
    g=[[] for _ in range(n+1)]
    for u,v in edges:
        g[u].append(v); g[v].append(u)
    res=[]
    def dfs(u,path,vis):
        if u==t: res.append(path[:]); return
        for v in g[u]:
            if v not in vis:
                vis.add(v)
                dfs(v,path+[v],vis)
                vis.remove(v)
    dfs(s,[s],{s})
    return res
```

Has Path (BFS)

Quick existence check between two nodes.

```
def has_path(n, edges, s, t):
    g=[[] for _ in range(n+1)]
    for u,v in edges:
        g[u].append(v); g[v].append(u)
    from collections import deque
    q=deque([s]); vis={s}
    while q:
        u=q.popleft()
        if u==t: return True
        for v in g[u]:
            if v not in vis:
                vis.add(v); q.append(v)
    return False
```

Cycle Detection

Use when: You must check if graph has loops (cycles) — important in dependency resolution, topological sort, or union-find problems.

Why it matters: - Dependency graphs: Cycles mean circular dependencies (impossible to resolve) - Task scheduling: Cycles prevent determining valid execution order - Deadlock detection: Cycles indicate resource deadlock in concurrent systems - Graph validity: Some algorithms require acyclic graphs (DAGs)

Undirected

Detects cycle via DFS.

Key idea: In undirected graphs, a cycle exists if we visit a node that's already visited AND it's not our immediate parent (where we came from). If we reach a visited node through a different path, we've found a cycle.

Example: Graph 1-2-3 with edge 3-1 has cycle: 1→2→3→1

```
def cycle_undirected(n, edges):
    g=[[] for _ in range(n+1)]
    for u,v in edges:
        g[u].append(v); g[v].append(u)
    vis=[0]*(n+1)
    def dfs(u,p):
        vis[u]=1
        for v in g[u]:
            if not vis[v]:
                if dfs(v,u): return True
            elif v!=p: return True
        return False
    for i in range(1,n+1):
        if not vis[i]:
            if dfs(i,-1): return True
    return False
```

Directed (Color DFS)

Detects cycle using node color states. Useful for topological sort problems.

Three-color algorithm: - **White (0):** Not visited yet — node hasn't been explored - **Gray (1):** Currently visiting — node is in current DFS path (recursion stack) - **Black (2):** Fully processed — node and all its descendants are done

Cycle detection: If we reach a GRAY node during DFS, we've found a back edge to an ancestor in the current path, meaning there's a cycle.

Key difference from undirected: In directed graphs, we need to track the current DFS path (gray nodes). Visiting a black node is fine (already processed), but gray means we're revisiting an ancestor = cycle!

Example: Graph: 1→2→3→1 has cycle (3 points back to 1 in same path) Graph: 1→2, 1→3, 2→4, 3→4 is OK (4 is visited twice but from different paths)

```python
def cycle_directed(n, edges):
    g=[[] for _ in range(n+1)]
    for u,v in edges: g[u].append(v)
    W,G,B=0,1,2; color=[W]*(n+1)
    def dfs(u):
        color[u]=G
        for v in g[u]:
            if color[v]==G: return True
            if color[v]==W and dfs(v): return True
        color[u]=B; return False
    for i in range(1,n+1):
        if color[i]==W:
            if dfs(i): return True
    return False
```

Data Structures

Use when: You need efficient query/update operations beyond simple arrays. Pick based on operation type: range queries, point updates, or special orderings.

Fenwick Tree (Binary Indexed Tree)

Use when: Need efficient prefix sums with point updates, or range sum queries. **Time:** Update $O(\log N)$, Query $O(\log N)$, Space $O(N)$ **Typical for:** Cumulative frequency, inversion counting, range sum queries with updates.

```python
class FenwickTree:
    def __init__(self, size):
        self.n = size
        self.tree = [0] * (self.n + 1)

    def update(self, index, delta):
        """Add delta to element at position index."""
        while index <= self.n:
            self.tree[index] += delta
            index += index & -index

    def query(self, index):
        """Return prefix sum from 1 to index."""
        res = 0
        while index > 0:
            res += self.tree[index]
            index -= index & -index
        return res

    def range_sum(self, left, right):
        """Return sum in range [left, right]."""
        return self.query(right) - self.query(left - 1)

# Usage Example
arr = [3, 2, -1, 6, 5, 4, -3, 3, 7, 2, 3]
n = len(arr)
bit = FenwickTree(n)

# Build tree (1-indexed)
for i in range(n):
    bit.update(i + 1, arr[i])

print(bit.query(5))           # prefix sum up to index 5
print(bit.range_sum(3, 8))    # sum in range [3, 8]
bit.update(4, 2)              # add 2 to arr[3]
print(bit.range_sum(3, 8))    # updated range sum
```

Segment Tree

Use when: Need range queries (min/max/sum/gcd) with range updates. **Time:** Build $O(N)$, Query/Update $O(\log N)$, Space $O(4N)$ **Typical for:** Range minimum/maximum query (RMQ), lazy propagation, interval problems.

What it is: A binary tree where each node stores aggregate information about a segment (range) of the array. Leaf nodes represent individual elements, internal nodes represent merged results of their children.

Key advantages over Fenwick Tree: - Supports ANY associative operation (min, max, gcd, etc.), not just sum - Can handle range updates efficiently with lazy propagation - Works on non-invertible operations (e.g., min/max don't have "inverse")

Tree structure: - Node at index i has children at $2i + 1$ (left) and $2i + 2$ (right) - Leaf nodes: $[i, i]$ for each array element - Internal nodes: $[l, r]$ where result = merge(left_child, right_child)

Common operations: 1. Point update: Change single element, update ancestors $O(\log N)$ 2. Range query: Get aggregate over $[l, r]$ by combining relevant segments 3. Range update: Update entire range (use lazy propagation for efficiency)

When to use vs Fenwick Tree: - Use Segment Tree: Need min/max/gcd, range updates, or complex operations - Use Fenwick Tree: Only need sum/prefix operations, simpler less memory

```
class SegmentTree:
    def __init__(self, arr):
        self.n = len(arr)
        self.tree = [0] * (4 * self.n)
        self.build(arr, 0, 0, self.n - 1)

    def build(self, arr, node, start, end):
        if start == end:
            self.tree[node] = arr[start]
        else:
            mid = (start + end) // 2
            self.build(arr, 2*node+1, start, mid)
            self.build(arr, 2*node+2, mid+1, end)
            self.tree[node] = self.tree[2*node+1] + self.tree[2*node+2]

    def update(self, node, start, end, idx, val):
        if start == end:
            self.tree[node] = val
        else:
            mid = (start + end) // 2
            if idx <= mid:
                self.update(2*node+1, start, mid, idx, val)
            else:
                self.update(2*node+2, mid+1, end, idx, val)
            self.tree[node] = self.tree[2*node+1] + self.tree[2*node+2]

    def query(self, node, start, end, l, r):
        if r < start or end < l:
            return 0
        if l <= start and end <= r:
            return self.tree[node]
        mid = (start + end) // 2
        return self.query(2*node+1, start, mid, l, r) + \
               self.query(2*node+2, mid+1, end, l, r)

# Usage
arr = [1, 3, 5, 7, 9, 11]
st = SegmentTree(arr)

# Query examples
print(st.query(0, 0, len(arr)-1, 1, 3))  # sum[1:3] = 3+5+7 = 15
print(st.query(0, 0, len(arr)-1, 0, 5))  # sum[0:5] = 1+3+5+7+9+11 = 36
print(st.query(0, 0, len(arr)-1, 2, 2))  # sum[2:2] = 5

# Update examples
st.update(0, 0, len(arr)-1, 1, 10)       # arr[1] = 10
print(st.query(0, 0, len(arr)-1, 1, 3))  # sum[1:3] = 10+5+7 = 22

st.update(0, 0, len(arr)-1, 0, 100)      # arr[0] = 100
```

```
print(st.query(0, 0, len(arr)-1, 0, 2))  # sum[0:2] = 100+10+5 = 115

# Multiple updates
st.update(0, 0, len(arr)-1, 3, 20)       # arr[3] = 20
st.update(0, 0, len(arr)-1, 4, 30)       # arr[4] = 30
print(st.query(0, 0, len(arr)-1, 3, 4))  # sum[3:4] = 20+30 = 50
```

Segment Tree with Lazy Propagation (Range Updates): Lazy propagation delays updates to children until necessary, making range updates $O(\log N)$ instead of $O(N)$.

Idea: Store pending updates in a separate lazy array. When visiting a node: 1. Apply any pending update from lazy array 2. If doing range update, update current node and mark children as lazy 3. Push down lazy values to children only when needed

```
class SegmentTreeLazy:
    def __init__(self, arr):
        self.n = len(arr)
        self.tree = [0] * (4 * self.n)
        self.lazy = [0] * (4 * self.n)  # pending updates
        self.build(arr, 0, 0, self.n - 1)

    def build(self, arr, node, start, end):
        if start == end:
            self.tree[node] = arr[start]
        else:
            mid = (start + end) // 2
            self.build(arr, 2*node+1, start, mid)
            self.build(arr, 2*node+2, mid+1, end)
            self.tree[node] = self.tree[2*node+1] + self.tree[2*node+2]

    def push(self, node, start, end):
        """Push lazy update down to children"""
        if self.lazy[node] != 0:
            self.tree[node] += (end - start + 1) * self.lazy[node]
            if start != end:  # not a leaf
                self.lazy[2*node+1] += self.lazy[node]
                self.lazy[2*node+2] += self.lazy[node]
            self.lazy[node] = 0

    def range_update(self, node, start, end, l, r, val):
        """Add val to all elements in range [l, r]"""
        self.push(node, start, end)
        if r < start or end < l:
            return
        if l <= start and end <= r:
            self.lazy[node] += val
            self.push(node, start, end)
            return
        mid = (start + end) // 2
        self.range_update(2*node+1, start, mid, l, r, val)
        self.range_update(2*node+2, mid+1, end, l, r, val)
        self.push(2*node+1, start, mid)
        self.push(2*node+2, mid+1, end)
        self.tree[node] = self.tree[2*node+1] + self.tree[2*node+2]

    def query(self, node, start, end, l, r):
        if r < start or end < l:
            return 0
        self.push(node, start, end)
        if l <= start and end <= r:
            return self.tree[node]
        mid = (start + end) // 2
        return self.query(2*node+1, start, mid, l, r) + \
               self.query(2*node+2, mid+1, end, l, r)

# Usage: Range updates
arr = [1, 2, 3, 4, 5]
st = SegmentTreeLazy(arr)
st.range_update(0, 0, 4, 1, 3, 10)  # add 10 to arr[1:4]
print(st.query(0, 0, 4, 0, 4))       # query sum of entire array
```

Variations: - **Min/Max Segment Tree:** Change merge operation to min() or max() - **GCD Segment Tree:** Use gcd() as merge function - **Count Segment Tree:** Count elements satisfying a condition in range

```
# Range Minimum Query (RMQ)
class MinSegmentTree:
    def __init__(self, arr):
        self.n = len(arr)
        self.tree = [float('inf')] * (4 * self.n)
        self.build(arr, 0, 0, self.n - 1)

    def build(self, arr, node, start, end):
        if start == end:
            self.tree[node] = arr[start]
        else:
            mid = (start + end) // 2
            self.build(arr, 2*node+1, start, mid)
            self.build(arr, 2*node+2, mid+1, end)
            self.tree[node] = min(self.tree[2*node+1], self.tree[2*node+2])

    def update(self, node, start, end, idx, val):
        if start == end:
            self.tree[node] = val
        else:
            mid = (start + end) // 2
            if idx <= mid:
                self.update(2*node+1, start, mid, idx, val)
            else:
                self.update(2*node+2, mid+1, end, idx, val)
            self.tree[node] = min(self.tree[2*node+1], self.tree[2*node+2])

    def query(self, node, start, end, l, r):
        if r < start or end < l:
            return float('inf')
        if l <= start and end <= r:
            return self.tree[node]
        mid = (start + end) // 2
        return min(self.query(2*node+1, start, mid, l, r),
                   self.query(2*node+2, mid+1, end, l, r))

# Usage: Min queries
arr = [2, 5, 1, 4, 9, 3]
min_st = MinSegmentTree(arr)
print(min_st.query(0, 0, 5, 2, 4))  # min[2:4] = 1
print(min_st.query(0, 0, 5, 0, 5))  # min[0:5] = 1
min_st.update(0, 0, 5, 2, 10)       # arr[2] = 10
print(min_st.query(0, 0, 5, 2, 4))  # min[2:4] = 4 (now)
```

```
# Range Maximum Query (RMQ)
class MaxSegmentTree:
    def __init__(self, arr):
        self.n = len(arr)
        self.tree = [float('-inf')] * (4 * self.n)
        self.build(arr, 0, 0, self.n - 1)

    def build(self, arr, node, start, end):
        if start == end:
            self.tree[node] = arr[start]
        else:
            mid = (start + end) // 2
            self.build(arr, 2*node+1, start, mid)
            self.build(arr, 2*node+2, mid+1, end)
            self.tree[node] = max(self.tree[2*node+1], self.tree[2*node+2])

    def update(self, node, start, end, idx, val):
        if start == end:
            self.tree[node] = val
        else:
            mid = (start + end) // 2
            if idx <= mid:
                self.update(2*node+1, start, mid, idx, val)
```

```
            else:
                self.update(2*node+2, mid+1, end, idx, val)
            self.tree[node] = max(self.tree[2*node+1], self.tree[2*node+2])

    def query(self, node, start, end, l, r):
        if r < start or end < l:
            return float('-inf')
        if l <= start and end <= r:
            return self.tree[node]
        mid = (start + end) // 2
        return max(self.query(2*node+1, start, mid, l, r),
                   self.query(2*node+2, mid+1, end, l, r))

# Usage: Max queries
arr = [3, 1, 7, 4, 2, 9]
max_st = MaxSegmentTree(arr)
print(max_st.query(0, 0, 5, 1, 4))  # max[1:4] = 7
print(max_st.query(0, 0, 5, 0, 5))  # max[0:5] = 9
max_st.update(0, 0, 5, 1, 15)       # arr[1] = 15
print(max_st.query(0, 0, 5, 1, 4))  # max[1:4] = 15 (now)
```

```
# Range GCD Query
from math import gcd

class GCDSegmentTree:
    def __init__(self, arr):
        self.n = len(arr)
        self.tree = [0] * (4 * self.n)
        self.build(arr, 0, 0, self.n - 1)

    def build(self, arr, node, start, end):
        if start == end:
            self.tree[node] = arr[start]
        else:
            mid = (start + end) // 2
            self.build(arr, 2*node+1, start, mid)
            self.build(arr, 2*node+2, mid+1, end)
            self.tree[node] = gcd(self.tree[2*node+1], self.tree[2*node+2])

    def update(self, node, start, end, idx, val):
        if start == end:
            self.tree[node] = val
        else:
            mid = (start + end) // 2
            if idx <= mid:
                self.update(2*node+1, start, mid, idx, val)
            else:
                self.update(2*node+2, mid+1, end, idx, val)
            self.tree[node] = gcd(self.tree[2*node+1], self.tree[2*node+2])

    def query(self, node, start, end, l, r):
        if r < start or end < l:
            return 0
        if l <= start and end <= r:
            return self.tree[node]
        mid = (start + end) // 2
        return gcd(self.query(2*node+1, start, mid, l, r),
                   self.query(2*node+2, mid+1, end, l, r))

# Usage: GCD queries
arr = [6, 9, 12, 18, 24]
gcd_st = GCDSegmentTree(arr)
print(gcd_st.query(0, 0, 4, 0, 2))  # gcd(6,9,12) = 3
print(gcd_st.query(0, 0, 4, 1, 4))  # gcd(9,12,18,24) = 3
print(gcd_st.query(0, 0, 4, 0, 4))  # gcd(6,9,12,18,24) = 3
gcd_st.update(0, 0, 4, 0, 12)       # arr[0] = 12
print(gcd_st.query(0, 0, 4, 0, 2))  # gcd(12,9,12) = 3
```

Disjoint Set Union (DSU / Union-Find)

Use when: Need to track connected components with dynamic edge additions. **Time:** Union/Find $O(\alpha(N)) \approx O(1)$, Space $O(N)$ **Typical for:** Dynamic connectivity, Kruskal's MST, cycle detection in undirected graphs.

```
class DSU:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # path compression
        return self.parent[x]

    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if px == py: return False
        # Union by rank
        if self.rank[px] < self.rank[py]:
            px, py = py, px
        self.parent[py] = px
        if self.rank[px] == self.rank[py]:
            self.rank[px] += 1
        return True

    def connected(self, x, y):
        return self.find(x) == self.find(y)

# Usage
dsu = DSU(5)
dsu.union(0, 1)
dsu.union(1, 2)
print(dsu.connected(0, 2))  # True
print(dsu.connected(0, 3))  # False
```

Monotonic Stack/Queue

Use when: Need to find next/previous greater/smaller element efficiently. **Time:** $O(N)$ for processing all elements, Space $O(N)$ **Typical for:** Histogram problems, sliding window maximum, stock span problems.

```
# Next Greater Element
def next_greater(arr):
    n = len(arr)
    result = [-1] * n
    stack = []  # stores indices
    for i in range(n):
        while stack and arr[stack[-1]] < arr[i]:
            result[stack.pop()] = arr[i]
        stack.append(i)
    return result

# Sliding Window Maximum using deque
from collections import deque
def sliding_window_max(arr, k):
    dq = deque()  # stores indices
    result = []
    for i in range(len(arr)):
        # Remove elements outside window
        while dq and dq[0] <= i - k:
            dq.popleft()
        # Remove smaller elements (not useful)
        while dq and arr[dq[-1]] < arr[i]:
            dq.pop()
        dq.append(i)
        if i >= k - 1:
            result.append(arr[dq[0]])
    return result
```

```
print(next_greater([4, 5, 2, 10, 8]))  # [5, 10, 10, -1, -1]
print(sliding_window_max([1,3,-1,-3,5,3,6,7], 3))  # [3,3,5,5,6,7]
```

Trie (Prefix Tree)

Use when: Need efficient prefix search, word storage, or autocomplete. **Time:** Insert/Search $O(L)$ where L is word length, Space $O(N \cdot L \cdot A)$ **Typical for:** Dictionary search, prefix matching, XOR maximum problems.

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end = True

    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end

    def starts_with(self, prefix):
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True

# Usage
trie = Trie()
trie.insert("apple")
print(trie.search("apple"))      # True
print(trie.search("app"))        # False
print(trie.starts_with("app"))   # True
```

Priority Queue (Heap)

Use when: Need to efficiently get min/max element with dynamic insertions. **Time:** Insert/Delete $O(\log N)$, Get-Min/Max $O(1)$, Space $O(N)$ **Typical for:** Dijkstra, A*, scheduling, k-th largest/smallest element.

```
import heapq

# Min-heap (default)
min_heap = []
heapq.heappush(min_heap, 5)
heapq.heappush(min_heap, 2)
heapq.heappush(min_heap, 8)
print(heapq.heappop(min_heap))  # 2

# Max-heap (negate values)
max_heap = []
heapq.heappush(max_heap, -5)
heapq.heappush(max_heap, -2)
heapq.heappush(max_heap, -8)
print(-heapq.heappop(max_heap))  # 8
```

```
# Heapify existing list
arr = [3, 1, 4, 1, 5]
heapq.heapify(arr)  # O(n)
print(heapq.nsmallest(3, arr))  # [1, 1, 3]
print(heapq.nlargest(2, arr))   # [5, 4]
```

Prefix & Suffix Algorithms

Use when: Need efficient cumulative computations, range queries, or pattern matching. Pick based on: static vs dynamic data, exact vs approximate matching needs.

Prefix Sum Array

Use when: Multiple range sum queries on static array, or 2D grid sums. **Time:** Build $O(N)$, Query $O(1)$, Space $O(N)$ **Typical for:** Subarray sum queries, cumulative frequency, range average.

```
# 1D Prefix Sum
def build_prefix(arr):
    n = len(arr)
    prefix = [0] * (n + 1)
    for i in range(n):
        prefix[i + 1] = prefix[i] + arr[i]
    return prefix

def range_sum(prefix, l, r):
    """Sum of arr[l:r+1] using 0-indexed l,r"""
    return prefix[r + 1] - prefix[l]

arr = [3, 2, -1, 6, 5, 4]
prefix = build_prefix(arr)
print(range_sum(prefix, 1, 3))  # sum of arr[1:4] = 2+(-1)+6 = 7

# 2D Prefix Sum
def build_2d_prefix(grid):
    m, n = len(grid), len(grid[0])
    prefix = [[0]*(n+1) for _ in range(m+1)]
    for i in range(1, m+1):
        for j in range(1, n+1):
            prefix[i][j] = grid[i-1][j-1] + prefix[i-1][j] + \
                           prefix[i][j-1] - prefix[i-1][j-1]
    return prefix

def query_2d(prefix, r1, c1, r2, c2):
    """Sum of submatrix from (r1,c1) to (r2,c2) inclusive (0-indexed)"""
    r1+=1; c1+=1; r2+=1; c2+=1
    return prefix[r2][c2] - prefix[r1-1][c2] - \
           prefix[r2][c1-1] + prefix[r1-1][c1-1]
```

Difference Array

Use when: Multiple range updates followed by final state query. **Time:** Build $O(N)$, Update $O(1)$, Reconstruct $O(N)$ **Typical for:** Batch range updates, interval addition problems.

```
def range_update(n, updates):
    """Apply range updates: each update is (l, r, val)"""
    diff = [0] * (n + 1)
    for l, r, val in updates:
        diff[l] += val
        diff[r + 1] -= val

    # Reconstruct array
    result = [0] * n
    curr = 0
    for i in range(n):
        curr += diff[i]
        result[i] = curr
```

```
    return result

# Add 10 to range [1,3], add 5 to range [2,4]
updates = [(1, 3, 10), (2, 4, 5)]
print(range_update(5, updates))  # [0, 10, 15, 15, 5]
```

KMP (Knuth-Morris-Pratt)

Use when: Pattern matching in strings, finding all occurrences. **Time:** Preprocess $O(M)$, Search $O(N)$, Space $O(M)$ where M is pattern length **Typical for:** String search, pattern counting, substring problems.

```
def compute_lps(pattern):
    """Compute Longest Proper Prefix which is also Suffix"""
    m = len(pattern)
    lps = [0] * m
    length = 0
    i = 1
    while i < m:
        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length
            i += 1
        else:
            if length != 0:
                length = lps[length - 1]
            else:
                lps[i] = 0
                i += 1
    return lps

def kmp_search(text, pattern):
    """Find all occurrences of pattern in text"""
    n, m = len(text), len(pattern)
    if m == 0: return []

    lps = compute_lps(pattern)
    result = []
    i = j = 0

    while i < n:
        if text[i] == pattern[j]:
            i += 1
            j += 1
        if j == m:
            result.append(i - j)
            j = lps[j - 1]
        elif i < n and text[i] != pattern[j]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1
    return result

text = "ababcababa"
pattern = "aba"
print(kmp_search(text, pattern))  # [0, 5, 7]
```

Z-Algorithm

Use when: Need length of longest substring starting at each position matching prefix. **Time:** $O(N)$, Space $O(N)$ **Typical for:** Pattern matching, palindrome problems, string matching variations.

```
def z_algorithm(s):
    """Z[i] = length of longest substring starting at i matching prefix"""
    n = len(s)
    z = [0] * n
    z[0] = n
    l, r = 0, 0
```

```
    for i in range(1, n):
        if i > r:
            l = r = i
            while r < n and s[r - l] == s[r]:
                r += 1
            z[i] = r - l
            r -= 1
        else:
            k = i - l
            if z[k] < r - i + 1:
                z[i] = z[k]
            else:
                l = i
                while r < n and s[r - l] == s[r]:
                    r += 1
                z[i] = r - l
                r -= 1
    return z

def pattern_match_z(text, pattern):
    """Find all occurrences using Z-algorithm"""
    combined = pattern + "$" + text
    z = z_algorithm(combined)
    m = len(pattern)
    return [i - m - 1 for i in range(m + 1, len(combined)) if z[i] == m]

print(z_algorithm("aabxaabxcaabxaabxay"))
print(pattern_match_z("ababcababa", "aba"))  # [0, 5, 7]
```

Suffix Array

Use when: Need sorted suffixes for multiple string queries. **Time:** Build $O(N \log^2 N)$ or $O(N \log N)$, Space $O(N)$ **Typical for:** Longest common substring, suffix-based string problems.

```
def build_suffix_array(s):
    """Build suffix array using sorting (simple version)"""
    n = len(s)
    suffixes = [(s[i:], i) for i in range(n)]
    suffixes.sort()
    return [suf[1] for suf in suffixes]

def lcp_array(s, suffix_arr):
    """Longest Common Prefix array"""
    n = len(s)
    rank = [0] * n
    for i in range(n):
        rank[suffix_arr[i]] = i

    lcp = [0] * n
    h = 0
    for i in range(n):
        if rank[i] > 0:
            j = suffix_arr[rank[i] - 1]
            while i + h < n and j + h < n and s[i + h] == s[j + h]:
                h += 1
            lcp[rank[i]] = h
            if h > 0:
                h -= 1
    return lcp

s = "banana"
sa = build_suffix_array(s)
print("Suffix Array:", sa)  # [5, 3, 1, 0, 4, 2]
print("Suffixes:", [s[i:] for i in sa])
lcp = lcp_array(s, sa)
print("LCP Array:", lcp)
```

Rolling Hash (Rabin-Karp)

Use when: Fast substring comparison, pattern matching with multiple patterns. **Time:** Preprocess $O(N)$, Query $O(1)$, Space $O(N)$ **Typical for:** Duplicate substring detection, substring equality checks.

```
class RollingHash:
    def __init__(self, s, base=31, mod=10**9 + 7):
        self.n = len(s)
        self.base = base
        self.mod = mod
        self.hash = [0] * (self.n + 1)
        self.pow = [1] * (self.n + 1)

        # Build hash and power arrays
        for i in range(self.n):
            self.hash[i + 1] = (self.hash[i] * base + ord(s[i])) % mod
            self.pow[i + 1] = (self.pow[i] * base) % mod

    def get_hash(self, l, r):
        """Get hash of substring s[l:r+1] (0-indexed)"""
        h = (self.hash[r + 1] - self.hash[l] * self.pow[r - l + 1]) % self.mod
        return h if h >= 0 else h + self.mod

# Usage
s = "abcabcabc"
rh = RollingHash(s)
print(rh.get_hash(0, 2) == rh.get_hash(3, 5))  # True: "abc" == "abc"
print(rh.get_hash(0, 2) == rh.get_hash(1, 3))  # False: "abc" != "bca"
```

Dynamic Programming Patterns

Use when: Problem has overlapping subproblems and optimal substructure. Pick based on: sequence optimization, counting paths, or state transitions.

Longest Increasing Subsequence (LIS)

Use when: Find longest strictly increasing subsequence in array. **Time:** $O(N^2)$ DP, $O(N \log N)$ Binary Search + DP **Typical for:** Sequence optimization, box stacking, Russian doll problems.

```
# O(N^2) Dynamic Programming Solution
def lis_dp(arr):
    """Returns length of LIS"""
    n = len(arr)
    if n == 0: return 0

    dp = [1] * n  # dp[i] = length of LIS ending at i

    for i in range(1, n):
        for j in range(i):
            if arr[j] < arr[i]:
                dp[i] = max(dp[i], dp[j] + 1)

    return max(dp)

# Get actual LIS sequence
def lis_sequence_dp(arr):
    n = len(arr)
    if n == 0: return []

    dp = [1] * n
    parent = [-1] * n

    for i in range(1, n):
        for j in range(i):
            if arr[j] < arr[i] and dp[j] + 1 > dp[i]:
                dp[i] = dp[j] + 1
                parent[i] = j
```

```
    # Reconstruct sequence
    max_len = max(dp)
    max_idx = dp.index(max_len)

    lis = []
    while max_idx != -1:
        lis.append(arr[max_idx])
        max_idx = parent[max_idx]

    return lis[::-1]

arr = [10, 9, 2, 5, 3, 7, 101, 18]
print(lis_dp(arr))  # 4
print(lis_sequence_dp(arr))  # [2, 3, 7, 18] or [2, 5, 7, 18]
```

```
# O(N log N) Binary Search Solution
def lis_binary_search(arr):
    """Returns length of LIS using binary search"""
    from bisect import bisect_left

    if not arr: return 0

    # tails[i] = smallest tail of all increasing subsequences of length i+1
    tails = []

    for num in arr:
        pos = bisect_left(tails, num)
        if pos == len(tails):
            tails.append(num)
        else:
            tails[pos] = num

    return len(tails)

# Get actual LIS sequence with binary search
def lis_sequence_binary(arr):
    from bisect import bisect_left

    n = len(arr)
    if n == 0: return []

    tails = []
    parent = [-1] * n
    indices = []  # indices[i] = index in arr for tails[i]

    for i, num in enumerate(arr):
        pos = bisect_left(tails, num)

        if pos == len(tails):
            tails.append(num)
            indices.append(i)
        else:
            tails[pos] = num
            indices[pos] = i

        if pos > 0:
            parent[i] = indices[pos - 1]

    # Reconstruct
    lis = []
    idx = indices[-1]
    while idx != -1:
        lis.append(arr[idx])
        idx = parent[idx]

    return lis[::-1]

arr = [10, 9, 2, 5, 3, 7, 101, 18]
print(lis_binary_search(arr))  # 4
print(lis_sequence_binary(arr))  # [2, 3, 7, 18]
```

Longest Decreasing Subsequence (LDS)

Use when: Find longest strictly decreasing subsequence. **Trick:** Negate all elements and find LIS, or reverse logic.

```
def lds(arr):
    """Longest Decreasing Subsequence"""
    # Method 1: Negate and find LIS
    negated = [-x for x in arr]
    return lis_binary_search(negated)

# Method 2: Modify LIS logic directly
def lds_direct(arr):
    from bisect import bisect_left

    tails = []
    for num in arr:
        # For decreasing: we want num < all in tails
        # Use bisect on negated values
        pos = bisect_left(tails, -num)
        if pos == len(tails):
            tails.append(-num)
        else:
            tails[pos] = -num

    return len(tails)

arr = [10, 9, 2, 5, 3, 7, 101, 18]
print(lds(arr))  # 2 (e.g., [10, 9] or [101, 18])
```

Longest Non-Decreasing Subsequence

Use when: Allow equal elements (non-strict inequality). **Trick:** Use 'bisect$_r ight$'$insteadof$'$bisect_l eft$'.

```
def lis_non_strict(arr):
    """LIS allowing equal elements (<=)"""
    from bisect import bisect_right

    tails = []
    for num in arr:
        pos = bisect_right(tails, num)  # Changed from bisect_left
        if pos == len(tails):
            tails.append(num)
        else:
            tails[pos] = num

    return len(tails)

arr = [1, 3, 3, 4, 2, 2, 5]
print(lis_non_strict(arr))  # 5: [1, 3, 3, 4, 5] or [1, 2, 2, 4, 5]
```

Number of LIS

Use when: Count how many different LIS exist. **Time:** $O(N^2)$ or $O(N \log N)$ with segment tree

```
def number_of_lis(arr):
    """Count number of longest increasing subsequences"""
    n = len(arr)
    if n == 0: return 0

    lengths = [1] * n  # lengths[i] = length of LIS ending at i
    counts = [1] * n   # counts[i] = number of LIS ending at i

    for i in range(1, n):
        for j in range(i):
            if arr[j] < arr[i]:
```

```
                if lengths[j] + 1 > lengths[i]:
                    lengths[i] = lengths[j] + 1
                    counts[i] = counts[j]
                elif lengths[j] + 1 == lengths[i]:
                    counts[i] += counts[j]

    max_len = max(lengths)
    return sum(c for l, c in zip(lengths, counts) if l == max_len)

arr = [1, 3, 5, 4, 7]
print(number_of_lis(arr))  # 2: [1,3,5,7] and [1,3,4,7]
```

LIS Variations & Applications

Box Stacking: Sort boxes, apply LIS on dimensions. **Russian Doll Envelopes:** Sort by width, LIS on height. **Maximum Sum Increasing Subsequence:** Track sum instead of length.

```
# Maximum Sum Increasing Subsequence
def max_sum_lis(arr):
    """Find max sum of increasing subsequence"""
    n = len(arr)
    if n == 0: return 0

    dp = arr[:]  # dp[i] = max sum of LIS ending at i

    for i in range(1, n):
        for j in range(i):
            if arr[j] < arr[i]:
                dp[i] = max(dp[i], dp[j] + arr[i])

    return max(dp)

arr = [1, 101, 2, 3, 100, 4, 5]
print(max_sum_lis(arr))  # 106: [1, 2, 3, 100]

# Russian Doll Envelopes (2D LIS)
def max_envelopes(envelopes):
    """Max number of envelopes that can fit inside each other"""
    from bisect import bisect_left

    # Sort by width asc, height desc (to handle equal widths)
    envelopes.sort(key=lambda x: (x[0], -x[1]))

    # Apply LIS on heights
    tails = []
    for _, h in envelopes:
        pos = bisect_left(tails, h)
        if pos == len(tails):
            tails.append(h)
        else:
            tails[pos] = h

    return len(tails)

envelopes = [[5,4],[6,4],[6,7],[2,3]]
print(max_envelopes(envelopes))  # 3: [2,3] -> [5,4] -> [6,7]
```

Bitmask Dynamic Programming

Use when: Need to track subset states, optimize over all subsets, or solve problems with small state space ($N \leq 20$). **Time:** Typically $O(2^N \cdot N)$ or $O(2^N \cdot N^2)$, Space $O(2^N)$ **Typical for:** Traveling Salesman, Assignment Problems, Subset Sum, State Compression.

Key Concepts:

- Use integers as bitmasks to represent subsets
- Bit i is set (1) if element i is included, unset (0) otherwise

- State: `dp[mask]` or `dp[mask][i]` where mask represents a subset
- Iterate through all subsets: `for mask in range(1 << n)`
- Iterate through submasks: `submask = (submask - 1) & mask`

Common Bit Operations:

```
# Check if i-th bit is set
if mask & (1 << i):  # True if bit i is 1

# Set i-th bit (add element i)
mask |= (1 << i)

# Clear i-th bit (remove element i)
mask &= ~(1 << i)

# Toggle i-th bit
mask ^= (1 << i)

# Count number of set bits (popcount)
bin(mask).count('1')

# Get lowest set bit
mask & (-mask)

# Remove lowest set bit
mask & (mask - 1)

# Iterate through all subsets of mask
submask = mask
while submask:
    # Process submask
    submask = (submask - 1) & mask
```

Problem 1: Traveling Salesman Problem (TSP) Find minimum cost to visit all cities exactly once and return to start.

```
def tsp(dist):
    """
    dist[i][j] = distance from city i to city j
    Returns minimum cost to visit all cities
    """
    n = len(dist)
    INF = float('inf')

    # dp[mask][i] = min cost to visit cities in mask, ending at city i
    dp = [[INF] * n for _ in range(1 << n)]
    dp[1][0] = 0  # Start at city 0

    # Iterate through all subsets
    for mask in range(1 << n):
        for last in range(n):
            if not (mask & (1 << last)):
                continue
            if dp[mask][last] == INF:
                continue

            # Try adding each unvisited city
            for nxt in range(n):
                if mask & (1 << nxt):  # Already visited
                    continue

                new_mask = mask | (1 << nxt)
                dp[new_mask][nxt] = min(dp[new_mask][nxt],
                                        dp[mask][last] + dist[last][nxt])

    # Return to starting city
    full_mask = (1 << n) - 1
    ans = min(dp[full_mask][i] + dist[i][0] for i in range(1, n))
    return ans
```

```
# Example
dist = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]
print(tsp(dist))  # 80
```

Problem 2: Assignment Problem Assign N tasks to N people, minimize total cost.

```
def assignment_problem(cost):
    """
    cost[i][j] = cost for person i to do task j
    Returns minimum total cost
    """
    n = len(cost)
    INF = float('inf')

    # dp[mask] = min cost to assign tasks in mask to first |mask| people
    dp = [INF] * (1 << n)
    dp[0] = 0

    for mask in range(1 << n):
        if dp[mask] == INF:
            continue

        person = bin(mask).count('1')  # Number of people assigned
        if person >= n:
            continue

        for task in range(n):
            if mask & (1 << task):  # Task already assigned
                continue

            new_mask = mask | (1 << task)
            dp[new_mask] = min(dp[new_mask], dp[mask] + cost[person][task])

    return dp[(1 << n) - 1]

# Example: 3 people, 3 tasks
cost = [
    [9, 2, 7],    # Person 0 costs
    [6, 4, 3],    # Person 1 costs
    [5, 8, 1]     # Person 2 costs
]
print(assignment_problem(cost))  # 13: person 0->task 1, person 1->task 2, person 2->task 0
```

Number Theory

- **Prime Numbers:** $n > 1$ divisible only by 1 and n. *Use:* For primality checks, factorization, or generating primes.

```
def is_prime(n):
    if n < 2: return False
    for i in range(2,int(n**0.5)+1):
        if n % i == 0: return False
    return True
```

- **Prime Factorization:** $n = p_1^{k_1} \dots p_m^{k_m}$ *Use:* For problems involving divisors, LCM/GCD, or number partitions.

```
def prime_factors(n):
    i = 2; factors = {}
    while i*i <= n:
        while n % i == 0:
```

```
            factors[i] = factors.get(i,0)+1
            n //= i
        i += 1
    if n > 1: factors[n] = 1
    return factors
```

- **GCD / LCM:** $\gcd(a,b)$, $\text{lcm}(a,b) = a*b//\gcd(a,b)$ *Use:* Simplifying fractions, divisibility problems, or LCM constraints.

```
from math import gcd
def lcm(a,b): return a*b//gcd(a,b)
```

- **Modulo Properties:** $(a+b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m$ $(a*b) \bmod m = ((a \bmod m) * (b \bmod m)) \bmod m$ *Use:* To avoid integer overflow and handle modular arithmetic.

- **Sieve of Eratosthenes:** Generate all primes $\leq N$. *Use:* Precompute primes for factorization or counting problems.

```
def sieve(n):
    is_prime=[True]*(n+1)
    is_prime[0]=is_prime[1]=False
    for i in range(2,int(n**0.5)+1):
        if is_prime[i]:
            for j in range(i*i,n+1,i):
                is_prime[j]=False
    return [i for i,p in enumerate(is_prime) if p]
```

- **Iterating Powers:** Compute $a^k \leq N$ efficiently. *Use:* For prime powers, factorization, and perfect powers.

```
def powers_up_to(a,N):
    p=a; res=[]
    while p<=N:
        res.append(p)
        p*=a
    return res
```

- **Arithmetic Progression (AP):** Sequence where difference between consecutive terms is constant. Formula: $a_n = a_1 + (n-1)d$ where d is common difference. Sum: $S_n = \frac{n}{2}(a_1 + a_n) = \frac{n}{2}(2a_1 + (n-1)d)$ *Use:* Problems with evenly spaced sequences, step counting, linear patterns.

```
# nth term: a1 + (n-1)*d
def ap_nth_term(a1, d, n):
    return a1 + (n - 1) * d

# Sum of first n terms
def ap_sum(a1, d, n):
    return n * (2*a1 + (n-1)*d) // 2

# Sum from a1 to an
def ap_sum_range(a1, an, n):
    return n * (a1 + an) // 2
```

- **Geometric Progression (GP):** Sequence where ratio between consecutive terms is constant. Formula: $a_n = a_1 \cdot r^{n-1}$ where r is common ratio. Sum: $S_n = a_1 \cdot \frac{r^n-1}{r-1}$ (if $r \neq 1$), $S_n = n \cdot a_1$ (if $r = 1$) Infinite sum (if $|r| < 1$): $S_\infty = \frac{a_1}{1-r}$ *Use:* Exponential growth/decay, compound interest, repeated doubling/halving.

```
# nth term: a1 * r^(n-1)
def gp_nth_term(a1, r, n):
    return a1 * (r ** (n - 1))

# Sum of first n terms
def gp_sum(a1, r, n):
```

```
    if r == 1: return n * a1
    return a1 * (r**n - 1) // (r - 1)

# Infinite sum (|r| < 1)
def gp_infinite_sum(a1, r):
    if abs(r) >= 1: return float('inf')
    return a1 / (1 - r)
```

Combinatorics / Logic

- **Counting sets avoiding duplicates:** Use 'set()' in Python. *Use:* To count unique outcomes or numbers.

```
S = set()
for x in range(1, N+1):
    S.add(x*x)
```

- **Summation of maximal modulo:** X mod $A[i] = A[i] - 1$ if X divisible by $A[i]$. *Use:* To quickly compute sum/max modulo values in problems.
- **Strategy-based logic:** First/second player choice problems. *Use:* Game theory problems, where you simulate options and pick safe moves.

Geometry Formulas

- **Triangle:** $A = \frac{1}{2} \cdot \text{base} \cdot \text{height}$
- **Rectangle:** $A = \text{width} \cdot \text{height}$
- **Square:** $A = \text{side}^2$
- **Circle:** $A = \pi r^2$, $C = 2\pi r$
- **Trapezoid:** $A = \frac{(a+b)}{2} \cdot h$
- **Parallelogram:** $A = \text{base} \cdot \text{height}$
- **Polygon (vertices (x_i, y_i)):** $A = \frac{1}{2} \left| \sum_{i=1}^{n} x_i y_{i+1} - x_{i+1} y_i \right|$ *Use:* Shoelace formula for arbitrary polygons.

Common ICPC Geometry Problem Patterns

- **Polygon + Point Queries:** *Use:* Check if a point is inside a polygon, compute area, perimeter, or intersections. **Formulas / Tips:** - Area: Shoelace formula

$$A = \frac{1}{2} \left| \sum_{i=1}^{n} x_i y_{i+1} - x_{i+1} y_i \right|$$

  - Point inside polygon: Ray casting or winding number method. - Perimeter: Sum of edge lengths.

- **Circle + Point / Tangent / Intersection:** *Use:* Distances from points to circle, line-circle intersection, tangent lines. **Formulas / Tips:** - Circle equation: $(x-h)^2 + (y-k)^2 = r^2$ - Distance from point (x_0, y_0) to circle center (h, k):

$$d = \sqrt{(x_0 - h)^2 + (y_0 - k)^2}$$

  - Tangent line from point outside: Use $(x-h)(x_0-h) + (y-k)(y_0-k) = r^2$

- **Triangle + Special Points:** *Use:* Problems requiring centroid, incenter, circumcenter, orthocenter. **Formulas / Tips:** - Centroid: average of vertices: $G = \left(\frac{x_1+x_2+x_3}{3}, \frac{y_1+y_2+y_3}{3}\right)$ - Incenter: intersection of angle bisectors - Circumcenter: intersection of perpendicular bisectors - Orthocenter: intersection of altitudes

- **Closest Pair / Farthest Pair:** *Use:* Min/max distance between points. **Formulas / Tips:** - Distance formula: $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ - Optimize with divide and conquer or convex hull for farthest pair.
- **Convex Hull + Diameter / Width:** *Use:* Problems involving outer boundary, max/min distances. **Formulas / Tips:** - Convex hull: Graham scan or Andrew's monotone chain $O(n \log n)$ - Diameter (farthest pair): Rotating calipers on convex hull - Width (minimum distance between parallel lines): Rotating calipers
- **Sweep Line + Segment Intersection:** *Use:* Detect collisions, count intersecting segments, line segments in plane. **Formulas / Tips:** - Sort events by x-coordinate (start/end of segment) - Use balanced BST (set) for active segments - Check neighbors only for intersections
- **Dynamic Geometry (Moving Points):** *Use:* Points moving over time, collision detection, moving convex hulls. **Formulas / Tips:** - Represent motion parametrically: $(x(t), y(t))$ - Use event-based simulation to handle key moments - Often combine sweep line + events for efficiency
- **Circle-Circle / Circle-Line Intersections:** *Use:* Problems with multiple circles, tangency, coverage. **Formulas / Tips:** - Circle-circle intersection: solve two circle equations simultaneously - Circle-line intersection: solve quadratic from line equation into circle formula
- **Area / Perimeter Optimization:** *Use:* Max area or min perimeter under constraints. **Formulas / Tips:** - Use geometric inequalities: AM-GM, triangle inequality - Convex hull or rotating calipers can help maximize area/diameter

Complexities Summary

Algorithm/Data Structure	**Time**
Binary Search (bisect)	$O(\log N)$
Binary Search on Answer	$O(\log N \cdot T)$
DFS/BFS Component Count	$O(N + M)$
Shortest Path (BFS)	$O(N + M)$
Dijkstra	$O((N + M) \log N)$
Bellman-Ford	$O(NM)$
Bridge Finding (Tarjan)	$O(N + M)$
Cycle Detection	$O(N + M)$
All Paths (DFS)	Exponential
Fenwick Tree (Update/Query)	$O(\log N)$
Segment Tree (Update/Query)	$O(\log N)$
DSU (Union/Find)	$O(\alpha(N))$
Monotonic Stack	$O(N)$
Trie (Insert/Search)	$O(L)$
Heap (Insert/Delete)	$O(\log N)$
Prefix Sum (Build/Query)	$O(N)$ / $O(1)$
KMP Pattern Matching	$O(N + M)$
Z-Algorithm	$O(N)$
Suffix Array (Build)	$O(N \log^2 N)$
Rolling Hash (Query)	$O(1)$
LIS (DP)	$O(N^2)$
LIS (Binary Search)	$O(N \log N)$
Number of LIS	$O(N^2)$
Bitmask DP (TSP)	$O(2^N \cdot N^2)$
Bitmask DP (Assignment)	$O(2^N \cdot N)$
Subset Iteration	$O(2^N)$