Core Greedy Patterns

1. Sort-then-Pick Pattern

Description

Sort elements by a specific key, then make greedy choices in one pass through the sorted array.

When to Use

- When optimal solution depends on processing elements in a specific order
- When local optimal choice at each step leads to global optimum
- Problems involving intervals, ratios, or priority-based selection

Core Technique

1. Identify the sorting key (finish time, ratio, weight, etc.)
2. Sort elements by this key
3. Scan once and make greedy selections
4. Maintain invariants during selection

Applications

Problem	Sorting Key	Greedy Choice
Activity Selection	End time (ascending)	Pick if start ≥ last_end
Fractional Knapsack	Value/weight ratio (descending)	Take highest ratio first
Coin Change (Canonical)	Denomination (descending)	Use largest coin possible
Assign Cookies	Appetite & size (ascending)	Match smallest feasible pairs
Non-overlapping Intervals	End time (ascending)	Shoot arrow at end point
Single Machine Scheduling	Processing time (ascending)	Process shortest job first

Template Code

```
// Generic sort-then-pick pattern
vector<Item> items; // your data structure
sort(items.begin(), items.end(), comparator); // sort by key
```

```
int result = 0;
State state; // maintain problem state
for (const auto& item : items) {
    if (greedyCondition(item, state)) {
        result += selectItem(item);
        updateState(state, item);
    }
}
return result;
```

2. Two-Pointer Pattern

Description

After sorting, use two pointers to find optimal pairings or matches between elements from opposite ends.

When to Use

- Problems involving pairing or matching elements
- When you need to consider combinations of smallest/largest elements
- Optimization problems where extremes work well together

Core Technique

1. Sort the array(s)
2. Initialize pointers at start and end (or two separate arrays)
3. Move pointers based on greedy criteria
4. Make decisions based on current pointer positions

Applications

Problem	Pointer Strategy	Movement Rule
Boats to Save People	Light + Heavy weights	If sum ≤ limit, move both; else move heavy
Assign Cookies	Children + Cookies	If cookie satisfies child, move both; else move cookie

Container With Most Water	Left + Right boundaries	Move pointer with smaller height
3Sum Closest	Fixed + Two pointers	Move based on sum comparison

Template Code

```
// Generic two-pointer pattern
sort(arr.begin(), arr.end());
int left = 0, right = arr.size() - 1;
int result = 0;
while (left < right) { // or left <= right for some problems
    if (condition(arr[left], arr[right])) {
        result += process(arr[left], arr[right]);
        left++;
        right--;
    } else if (arr[left] + arr[right] < target) {
        left++;
    } else {
        right--;
    }
}
return result;
```

3. Heap-Based Pattern

Description

Use priority queues (heaps) to always access the optimal next choice efficiently.

When to Use

- When you need to repeatedly select minimum/maximum element
- Problems involving merging or combining elements optimally
- Scheduling problems with priorities

Core Technique

1. Initialize heap with appropriate elements

2. Repeatedly extract optimal element
3. Process the element and possibly add new elements
4. Continue until heap is empty or goal is reached

Applications

Problem	Heap Type	Extract Strategy
Connect Ropes	Min-heap	Merge two smallest, add result back
Job Sequencing	Min-heap	Keep profits, remove smallest if over capacity
Huffman Coding	Min-heap	Merge two minimum frequencies
Dijkstra's Algorithm	Min-heap	Extract minimum distance vertex
Kth Largest Element	Max-heap or Min-heap	Extract k times or maintain k-size heap

Template Code

```
// Generic heap-based pattern
priority_queue<int, vector<int>, greater<int>> minHeap; // min-heap
// or priority_queue<int> maxHeap; for max-heap

// Initialize heap
for (const auto& item : items) {
    minHeap.push(item);
}

int result = 0;
while (!minHeap.empty() && !goalReached()) {
    auto optimal = minHeap.top();
    minHeap.pop();

    result += process(optimal);

    // Possibly add new elements
    if (shouldAddMore()) {
        minHeap.push(newElement);
    }
}
```

```
return result;
```

4. Sweep Line Pattern

Description

Process events in chronological order, maintaining running state to track optimal values.

When to Use

- Problems involving intervals or time-based events
- When you need to track overlapping or concurrent activities
- Resource allocation problems

Core Technique

1. Create events from input (arrivals, departures, etc.)
2. Sort events by time
3. Process events maintaining current state
4. Track optimal value throughout the sweep

Applications

Problem	Events	State Tracking
Minimum Platforms	Arrivals/Departures	Current trains, max platforms
Meeting Rooms	Start/End times	Active meetings count
Gas Station	Cumulative surplus	Current tank, valid start
Skyline Problem	Building start/end	Active building heights

Template Code

```
// Generic sweep line pattern
vector<Event> events;
// Create events from input
```

```
for (const auto& interval : intervals) {
    events.push_back({interval.start, START});
    events.push_back({interval.end, END});
}

sort(events.begin(), events.end());

int currentState = 0, optimalValue = 0;
for (const auto& event : events) {
    if (event.type == START) {
        currentState++;
        optimalValue = max(optimalValue, currentState);
    } else {
        currentState--;
    }
}
return optimalValue;
```

5. Stack-Based Pattern

Description

Use a stack to maintain monotonic properties or optimal structures while processing elements.

When to Use

- Problems requiring lexicographic optimization
- When you need to maintain monotonic sequences
- Problems where later elements can improve earlier choices

Core Technique

1. Process elements from left to right
2. Maintain stack with specific properties (monotonic, etc.)
3. Pop elements when better choice is found
4. Push current element after processing

Applications

Problem	Stack Property	Pop Condition
Remove K Digits	Monotonic increasing	Stack top > current digit
Largest Rectangle	Monotonic increasing heights	Can't extend rectangle
Next Greater Element	Pending elements	Found greater element
Valid Parentheses	Balanced structure	Matching closing bracket

Template Code

```
// Generic stack-based pattern (Remove K Digits example)
string removeKDigits(string num, int k) {
    string stack;
    for (char digit : num) {
        while (k > 0 && !stack.empty() && stack.back() > digit) {
            stack.pop_back();
            k--;
        }
        stack.push_back(digit);
    }

    // Handle remaining k
    while (k > 0 && !stack.empty()) {
        stack.pop_back();
        k--;
    }

    return stack.empty() ? "0" : stack;
}
```

6. Range Extension Pattern

Description

Greedily extend current range or window to optimal boundary, then process.

When to Use

- Problems involving reachability or coverage
- When you need to process elements in layers or levels
- Segmentation problems with optimal boundaries

Core Technique

1. Track current range/window boundary
2. Extend boundary based on elements processed
3. When boundary is reached, finalize current segment
4. Start new segment if needed

Applications

Problem	Range Tracking	Extension Rule
Jump Game I	Farthest reachable	max(far, i + nums[i])
Jump Game II	Current level end	Extend to farthest when level complete
Partition Labels	Segment boundary	Extend to last occurrence of characters
Gas Station	Valid start position	Reset when tank < 0

Template Code

```
// Generic range extension pattern (Jump Game II)
int jump(vector<int>& nums) {
    int jumps = 0, currentEnd = 0, farthest = 0;

    for (int i = 0; i < nums.size() - 1; i++) {
        farthest = max(farthest, i + nums[i]);

        if (i == currentEnd) { // reached end of current range
            jumps++;
            currentEnd = farthest; // extend to new range
        }
    }

    return jumps;
}
```

7. Counter-Based Pattern

Description

Maintain counters or state variables to ensure local feasibility at each step.

When to Use

- Problems with resource constraints
- When you need to track availability of items
- Feasibility checking problems

Core Technique

1. Initialize counters for available resources
2. For each operation, check feasibility
3. Update counters based on greedy choice
4. Return false/impossible if constraints violated

Applications

Problem	Counters	Feasibility Check
Lemonade Change	Bills of each denomination	Can make required change
Task Scheduler	Character frequencies	Maintain minimum intervals
Hand of Straights	Card counts	Can form consecutive groups

Template Code

```
// Generic counter-based pattern (Lemonade Change)
bool lemonadeChange(vector<int>& bills) {
    int five = 0, ten = 0;

    for (int bill : bills) {
        if (bill == 5) {
            five++;
```

```
        } else if (bill == 10) {
            if (five == 0) return false;
            five--;
            ten++;
        } else { // bill == 20
            if (ten > 0 && five > 0) {
                ten--;
                five--;
            } else if (five >= 3) {
                five -= 3;
            } else {
                return false;
            }
        }
    }

    return true;
}
```

8. Graph-Based Greedy Pattern

Description

Apply greedy choices in graph algorithms, typically with supporting data structures.

When to Use

- Graph optimization problems (MST, shortest paths)
- When local optimal choices lead to global optimum in graphs
- Problems with cut properties or optimal substructure

Core Technique

1. Initialize supporting data structures (DSU, priority queue)
2. Process edges/vertices in optimal order
3. Make greedy choices based on graph properties

4. Use data structures to maintain efficiency

Applications

Problem	Data Structure	Greedy Choice
Kruskal MST	DSU + Sorted edges	Add lightest edge without cycle
Dijkstra	Priority queue	Process nearest unvisited vertex
Prim MST	Priority queue	Add lightest edge from current tree

Template Code

```
// Generic graph greedy pattern (Kruskal MST)
struct DSU {
    vector<int> parent, rank;
    DSU(int n) : parent(n), rank(n, 0) {
        iota(parent.begin(), parent.end(), 0);
    }

    int find(int x) {
        return parent[x] == x ? x : parent[x] = find(parent[x]);
    }

    bool unite(int x, int y) {
        x = find(x); y = find(y);
        if (x == y) return false;
        if (rank[x] < rank[y]) swap(x, y);
        parent[y] = x;
        if (rank[x] == rank[y]) rank[x]++;
        return true;
    }
};

int kruskal(int n, vector<tuple<int,int,int>>& edges) {
    sort(edges.begin(), edges.end()); // sort by weight
    DSU dsu(n);
    int mstWeight = 0, edgesUsed = 0;

    for (auto [w, u, v] : edges) {
        if (dsu.unite(u, v)) {
            mstWeight += w;
```

```
        if (++edgesUsed == n - 1) break;
      }
    }

    return mstWeight;
}
```

Pattern Selection Guide

Quick Decision Tree

1. **Need to process in specific order?** → Sort-then-Pick
2. **Need to pair/match elements?** → Two-Pointer
3. **Need repeated min/max selections?** → Heap-Based
4. **Processing time-based events?** → Sweep Line
5. **Need lexicographic optimization?** → Stack-Based
6. **Working with ranges/segments?** → Range Extension
7. **Resource/feasibility constraints?** → Counter-Based
8. **Graph optimization problem?** → Graph-Based Greedy

Complexity Analysis by Pattern

Pattern	Typical Time Complexity	Space Complexity
Sort-then-Pick	O(n log n)	O(1)
Two-Pointer	O(n log n)	O(1)
Heap-Based	O(n log n)	O(n)
Sweep Line	O(n log n)	O(n)
Stack-Based	O(n)	O(n)
Range Extension	O(n)	O(1)
Counter-Based	O(n)	O(k) where k is counter types
Graph-Based	O(E log E) or O(V log V)	O(V)

Success Factors

1. **Identify the pattern** - Recognize which core pattern applies
2. **Prove greedy choice** - Ensure local optimum leads to global optimum
3. **Choose right data structure** - Use appropriate supporting structures
4. **Handle edge cases** - Empty inputs, boundary conditions
5. **Verify time complexity** - Ensure solution fits within time limits