Graph traversal

BFS: Breadth First Search (1091/542/752/1306/286)

- Unweighted graph, find shortest path (min number of edges) from a given start node to a finish node (or all nodes)
- BFS: outward level-by-level traversal strategy (queue/FIFO), finds shortest path in unweighted graphs by # of steps (hops)
- Used in social networks, routing systems, friend suggestion
- Answers the following questions:
  - Can we reach node X?
  - How many hops to reach X?
  - What is the path to reach X?
  - Reach all nodes + used for grid fill (flood fill, 01 matrix, ...)
- Data structures:
  - queue []C or queue of (node, dist), dist = # hops from start
  - visited map[C]bool
  - optional: parent map[C]C for path reconstruction
- Initialize queue with start node and mark node as visited
- While queue not empty:
  - Pop node, skip if already visited, else mark as visited
  - Check if node is what we need → return node/dist
  - For each neighbor, if unvisited
    - enqueue & increment dist by 1 or track parent path
- Works on both directed and undirected graphs; handles cycles via visited check; If disconnected → some nodes not visited
- Time: O(V + E) – each node and edge is processed once
- Space: O(V) – visited + queue – O(V) size in worst case

DFS: Depth-First Search (547/200/207/210/417)

- Graph traversal: explore as deep as possible before backtracking
- Used to explore all nodes in a connected component or recursively traverse a graph-like structure
- Works on directed and undirected graphs (may be disconnected); handles cycles using visited array/map
- Step 1: build adjacency list from input (e.g. matrix or edge list): adj := make[][]int, n)
- Step 2: declare visited := make[]bool, n) and var dfs func(int) closure
- Step 3: for each node i in 0..n-1, if !visited[i] → dfs(i) and do additional logic e.g. increment component counter and so on...
- DFS logic: dfs(node) + mark visited[node] = true → for each neighbor in adj[node], dfs(neighbor)
- Each full DFS traversal = one connected component (e.g. one province)
- Often used to count provinces (LC 547), detect cycles, generate post-order for topo sort
- Also used in implicit graphs (e.g. recursion/decision trees): see Backtrack DFS section
- Related problems: LC 547 (Provinces), LC 207/210 (Course Schedule – Topo Sort), LC 200 (Islands), LC 417 (Water Flow)
- LC 547 (Number of provinces) can also be solved using Union-Find or BFS — see related cheat sheet sections
- DFS vs BFS: both explore reachable nodes; DFS uses recursion call stack (go deep), BFS uses queue (explore immediate neighbors level-by-level)
- DFS-based Topo Sort: post-order traversal + reverse result; see Topological Sort section

Dijkstra's Algorithm (743/1631/1514/787)

- Weighted graph with **non-negative** edge weights, find **shortest path** from a given start node to a finish node (or all nodes)
- Dijkstra is **greedy** traversal strategy, *pointly based on BFS* logic but with edge weights & priorities → expand to lowest cost node
- Used in GPS navigation, network routing, game AI path finding
- Data structures
  - Edge struct {To String, Weight int} to represent the graph
  - Item struct {Node string, Distance int} → entry for min heap
  - dist map[string]int → node's shortest distance from start
  - heap *min-heap of Item struct (Node string, Distance int)
  - visited map[string]bool → avoid reprocessing
  - parents map[string]string → reconstruct path (optional)
- heap.Interface(h[i].Distance < h[j].Distance)
- for heap.Len() > 0 {all till end in loop}
- heap.Pop() + get closest unprocessed node
- check if node was already visited and check if it's finish node
- For each neighbor:
  - Compute newDist = dist[curr] + edge weight
  - If newDist < dist[neighbor]: update dist and insert into heap
- O(V + E) log V) → pop and insert are "independent", hence +
  - insert and pop both cost O(log V)
  - pop once per node → O(V log V)
  - insert for each edge → O(E log V)

Lowest common ancestor (235)

- Goal: Find lowest common ancestor of a and b
- Recursion: lca(node, a, b) *Node
- If both values are smaller than curr node → LCA must be in the left subtree, same for bigger right tree
- if a < node && b < node → go left
- if a > node && b > node → go right
- else → return node - if values split (a < curr < b) or match curr (a = curr or b = curr) → curr is LCA)
- Time: O(h), h = height of tree

Trim BST (669)

- Goal: Remove nodes outside [min, max] range
- Recursion: getKth(node, *k) (int, bool)
- Recursion: trimBst(node, min, max) *Node
- If node == nil → return nil
- if node < min → recurse right
- if node > max → recurse left
- node.Left = trimBst(node.Left, min, max)
- node.Right = trimBst(node.Right, min, max)
- return node
- Time: O(n), Space: O(h), h = height of tree

Validate BST (98)

- Goal: Check if tree is valid BST
- Recursion: isValid(node, min, max) bool
- if node == nil → return true
- if node is out of bounds - return false, if not → recursively validate both left and right with updated bounds:
- if n.Value ≤ min or ≥ max → return false
- return isValid(n.Left, min, n.Value) && isValid(n.Right, n.Value, max)
- Time: O(n), Space: O(h), h = height of tree

Kth smallest element (230)

- Goal: Return kth smallest element
- Recursion: getKth(node, *k) (int, bool)
- if node == nil → return -1, false
- if found in left → return early
- *k--
- if *k == 0 → return node.Value, true
- recurse right
- Time: O(k) best, O(n) worst | Space: O(h)

Binary Search Trees (BST)

Core Idea (used by all problems)

- BST is a binary tree where:
  - Left subtree < node.Value < Right subtree
- In-order traversal (Left → Node → Right) returns elements in sorted order
- In all BST problems you always rely on BST invariant and on recursion (you recurse either left or right depending on some condition)
- For most BST problems, space complexity is:
- O(h) for recursion stack, where h is height of tree

Range in BST (938)

- Goal: Sum of values within range [low, high]
- Recursion: sumNodes(node, low, high) int
- if node == nil → return 0
- if node.Value < low → return sumNodes(n.Right, low, high)
- if node.Value > high → return sumNodes(n.Left, low, high)
- return node.Value + sumNodes(n.Left, low, high) + sumNodes(n.Right, low, high)
- Time: O(n) worst, O(log n) avg

String problems

Longest Palindromic Substring (5)

- Given a string, return the longest substring that is a palindrome
- A substring is a contiguous section of the string
- A palindrome reads the same forward and backward
- For each index i in the string:
  - Expand odd-length: expandFromCenter(str, i, i)
  - Expand even-length: expandFromCenter(str, i, i+1)
  - Track and return the longest found
- Expansion:
  - While s[left] == s[right], expand outwards: left--, right++
  - Final palindrome is s[left+1:right] (inclusive bounds)
- Time: O(n²) → 2n centers × expansion takes up to O(n)
- Space: O(1) → no extra memory except variables

Longest Common Prefix (14)

- Given an array of strings, return the longest common prefix
- If no common prefix exists, return ""
- Start with the first word as the prefix
- For each string s in the array (excluding the first):
  - While s does not start with prefix, trim the last character
  - Stop early if prefix becomes empty
- Return prefix
- Time: O(S), S is the total number of characters across all strings
- Space: O(1) extra (excluding input)
- Ofcourse, you can also solve this problem with trie

Group anagrams (49)

- Given a list of strings, group them so that each group contains words that are anagrams of each other
- Anagrams = words made of the same characters in different order
- Example: "tea", "eat", "ate" → same group
- Strategy (sort and group), for each word:
- Convert it to []rune and sort → gives a normalized "signature"
- Use the sorted string as the key in a map
- Append the original word to that group
- Return all values from the map
- Time: O(n * k log k) → n = number of words, k = avg word length
- Space: O(n*k) → n groups with up to k characters per key/value

Longest Substring Without Repeating (3)

- Given a string s, return the length of the longest substring without repeating characters
- A substring must be contiguous, and characters must be unique
- Use left (= 0 at beginning) and right (ds in the loop) pointers to define the current window
- Use a map[rune]int to track the last seen index of each char
- Move right forward: for right, c := range s
- If c is already in window (exists in map & its last seen idx >= left) → move left to lastSeenIdx + 1 to exclude the duplicate
- Always update
  - winLen = right - left + 1
  - maxLen = winLen, if winLen > maxLen
- Update lastSeen[c] = right on every step
- Time: O(n); Space: O(k) → where k is number of unique chars

Check anagram

```
// checks if t is an anagram of s
func isAnagram(s, t string) bool {
    if len(s) != len(t) {
        return false
    }
    freqMap := make(map[rune]int)
    for _, c := range s {
        freqMap[c]++
    }
    for _, c := range t {
        freqMap[c]--
        if freqMap[c] == 0 {
            delete(freqMap, c)
        } else if freqMap[c] < 0 {
            return false
        }
    }
    return len(freqMap) == 0
}
```

Check if string is palindrome

```
func checkPalindrome(str string) bool {
    left := 0
    right := len(str) - 1
    for left <= right {
        if str[left] != str[right] {
            return false
        }
        left++
        right--
    }
    return true
}
```

Topological sort (207/210/269/1203)

General

- Given n and edges [][]int, return one valid topological order (or detect cycle)
- Works on DAGs (Directed Acyclic Graphs); if cycle exists → return empty/error
- It's used in scheduling problems, dependency resolution and cycle detection
- We have 2 solutions for this problem: Kahn's algo and DFS-based topo sort
- They both
  - give correct topological orders (if graph is a DAG)
  - detect cycles (in different ways)
  - same time/space complexity: O(n + e)
  - solve same LC problems: 207/210/269/1203
- Prefer Kahn when tasks must be executed as soon as they're ready or if you are already computing in-degrees for another purpose
- Prefer DFS when you prefer recursive flow or when you want more localized and stronger cycle detection - easy to reconstruct the cycle path

Kahn's Algorithm

- Data structures:
  - adj [][]int – adjacency/neighbors list (ex. node 0 → [node 1, node 3])
  - inDegree []int – number of incoming edges per node (# of dependencies)
  - queue []int – ready to process nodes (nodes with inDegree[node] == 0)
  - result []int – final topological order
- Initialize the adj list and inDegree using length n, it ensures we loop through all possible nodes, including those that may not appear as sources in the edges
- Process edges - build adj list and increment count for inDegree array
- Initialize queue with all nodes where inDegree[node] == 0
- While queue not empty:
  - Pop a node, append to result
  - For each neighbor (use adj list) → decrement its in-degree by 1
  - If inDegree[neighbor] == 0, push neighbor node to the queue
- After processing:
  - If result contains all n nodes - len(result) == n) → valid topological order
  - Else → graph contains a cycle - some node(s) didn't reach inDegree = 0
- Time: O(n + e) – build graph + process each node and edge once
- Space: O(n + e) – adjacency list, in-degree array, queue, result
- BFS-style: nodes are explored level-by-level through immediate neighbors; node may be discovered early, but processed later when in-degree hits 0

DFS-based

- Data structures:
  - adj [][]int – adjacency/neighbors list (ex. node 0 → [node 1, node 3])
  - visited []int – 0 = unvisited, 1 = visiting, 2 = visited
  - result []int – stores nodes in post-order (we reverse at end)
- Build adj list from edges
- define helper closure function: dfs(node) bool (returns false if there is cycle)
  - If visited[node] == 1 → cycle detected (node is already on current DFS path)
  - If visited[node] == 2 → node already processed, return true
  - Mark node as visiting + visited[node] = 1
  - Recurse on all neighbors using adj list
  - If any dfs(neighbor) returns false → propagate return
  - After all neighbors processed, mark node as visited, append to result
- For each node: (makes sure to cover all nodes in disconnected graph)
  - If visited[node] == 0 → call dfs(node), propagate false return
- After dfs calls, reverse result to get valid topological order
- Time: O(n + e) – build graph + visit each node and edge once
- Space: O(n + e) – adjacency list, visited array, result, recursion stack
- DFS-style: nodes are explored via recursive depth-first traversal; result is built in post-order; cycle is detected by revisiting a node in the same DFS path

Greedy

Classroom Scheduling (435/452/1346/252/253)

- Schedule the max number of non-overlapping classes
- Greedy pick class that ends earliest & doesn't overlap
- Sort list of classes by End time
- for: if none picked (l is not overlapping → pick
- Time: O(n log n) → for sorting the slice
- Alternative approach: map + scan/delete → O(n²)

Set Cover Approximation

- Problem type: NP-hard — no known polynomial-time optimal solution
- Goal: cover a required set using the fewest number of subsets (e.g. radio stations covering US states)
- Brute force: exponential time (2^subsets × O(2^) → optimal but intractable
- Greedy approximation strategy can find an approximate solution
- While required set not empty:
  - Pick the subset that covers the most uncovered elements
  - Add it to result
  - Remove newly covered elements from required set
- Time: O(m × n), where m = required items, n = number of subsets
- Set operations:
  - intersection(a, b) → newly covered items
  - difference(a, b) → update uncovered set
- Doesn't guarantee optimal solution, but gives good results efficiently
- Applies to coverage/resource allocation problems where the optimal solution is too slow; Related patterns: greedy partitioning, range coverage, max gain per step strategies

Union find

Kruskal min span tree 1135/1584

```
- connected[undirected[weighted graph, nodes
0..n-1, edges [u, v, w], w is weight; return
min total weight for set of edges that connect
all nodes without forming cycles
func getMstCost(edges [][]int, n int) int {
    totalCost := 0
    sort.Slice(edges, func(i, j int) bool {
        return edges[i][2] < edges[j][2]
    }) // Sort edges by increasing weight
    uf := union_find.NewUnionFind[int]()
    for _, edge := range edges {// O(E*α(N))
        u, v, w := edge[0], edge[1], edge[2]
        if uf.Union(u, v) {
            totalCost += w
        }
    }
    return totalCost
}
```

Number of provinces 547 (also solved with BFS/DFS)

```
Given an n x n matrix `isConnected`, where:
- `isConnected[i][j] == 1` person i connected to person j
- `isConnected[i][j] == 0` means no direct connection
- Persons are labeled from 0 to n-1
- Return the number of provinces, where a province is a
group of directly or indirectly connected people.
func getNumProvinces(isConnected [][]int) int {
    n := len(isConnected)
    numProvinces := n
    uf := union_find.NewUnionFind[int]()
    for i := 0; i < n; i++ { // Iterate only ↰
        for j := i + 1; j < n; j++ {
            if isConnected[i][j] == 1 && uf.Union(i, j) {
                numProvinces--
            }
        }
    }
    return numProvinces
} // O(E*2 * α(n))
```

Basic structures

```
type UnionFind[T comparable] struct { 5 usages
    parent map[T]T
    rank   map[T]int
}
func NewUnionFind[T comparable]() *UnionFind[T] {
    return &UnionFind[T]{
        parent: make(map[T]T),
        rank:   make(map[T]int),
    }
}
func (uf *UnionFind[T]) Find(x T) T {
    if _, ok := uf.parent[x]; !ok {
        uf.Add(x)
    }
    if uf.parent[x] != x {
        uf.parent[x] = uf.Find(uf.parent[x])
    }
    return uf.parent[x]
}
func (uf *UnionFind[T]) Union(x, y T) bool { 1 usage
    rootX := uf.Find(x)
    rootY := uf.Find(y)
    if rootX == rootY {
        return false
    }
    if uf.rank[rootX] < uf.rank[rootY] {
        uf.parent[rootX] = rootY
    } else if uf.rank[rootX] > uf.rank[rootY] {
        uf.parent[rootY] = rootX
    } else {
        uf.parent[rootX] = rootY
        uf.rank[rootY]++
    }
    return true
}
func (uf *UnionFind[T]) Add(x T) {
    if _, ok := uf.parent[x]; !ok {
        uf.parent[x] = x
        uf.rank[x] = 0
    }
}
```

Count connected components 323/1319

```
func countConnectedComponents(n int, edges [][]int) int {
    uf := union_find.NewUnionFind[int]()
    componentCount := n
    for _, edge := range edges {
        u, v := edge[0], edge[1]
        if uf.Union(u, v) { // same as Find(u) != Find(v)
            componentCount--
        } // two components merged → count decreases
    }
    return componentCount
}
```

Cycle Detection

```
- undirected graph with n nodes, list
of edges, graph represents a cycle?
func containsCycle(edges [][]int) bool {
    uf := union_find.NewUnionFind[int]()
    for _, edge := range edges {
        u, v := edge[0], edge[1]
        if !uf.Union(u, v) {
            return true // Found a cycle
        }
    }
    return false // No cycles found
} // O(E * α(n)), E = # of edges
```

Accounts merge 721

- Group accounts by shared emails using Union-Find
- emails = nodes, shared email → union
- use generic UF (map-based) to avoid ID/email mapping
- emailToName map[string]string holds names per email
- first pass:
  - union all emails in same account/name
  - fill emailToName map
- second pass - loop emailToName
  - group by root: map[string][]string{ root → [emails]}
  - for each email Find root and append email to slice
- third pass - loop the map from previous step
  - build result: name + sorted group
- result: sorted list of [name, emails...]
- time: UF-O(E * α(E)) + sort O(E log E)
- space: O(E)

DP

Longest common subsequence

LCS Length (1143)

- DP grid dp[m+1][n+1]
- If A[i-1] == B[j-1] → dp[i][j] = dp[i-1][j-1] + 1
- Else → dp[i][j] = max(dp[i-1][j], dp[i][j-1])
- Final result is at: dp[m][n]
- Time: O(mn); Space: O(m*n)

LCS Reconstruction (1143)

- Walk backward from dp[m][n]
- If chars match → go ↖
  - add A[i-1], (i--, j--)
- Else → follow previous best LCS
  - if dp[i-1][j] > dp[i][j-1] (i--) else (j--)
- Reverse result

Min Inserts/Dels to convert A → B (583)

- LCSLen = lcs(A, B)
- # Deletions = len(A) - LCSLen
- # Insertions = len(B) - LCSLen

Shortest Common Supersequence (1092)

- Shortest string containing both A & B as subsequences
- SCS Length = len(A) + len(B) - LCS
- Reconstruction:
  - Loop LCS chars
  - Append A[i] & B[j] up to LCS char
  - Append LCS char once
  - Append remaining A[i:], B[j:]

Knapsack

0/1 Knapsack DP (416/474/1049)

- dp[i][j] = max value using first i items and sub-capacity j
- If item fits in sub-knapsack (item.Weight <= j) take max of sub-knapsack value if we take the item vs previous best:
  - newBest := item.Value + dp[i-1][j-item.Weight]
  - previousBest := dp[i-1][j]
  - dp[i][j] = maximt(previousBest, newBest)
- Else take previous best: dp[i][j] = previousBest
- Solution is at: dp[n][capacity]
- Reconstruct items → i = n; j = capacity; for i > 0:
  - If dp[i][j] != dp[i-1][j] → item[i-1] taken
    - move j -= item[i-1].Weight
  - move to previous item in any case: i--
- Time: O(n * capacity)
- Space: O(n * capacity), can reduce to O(capacity) (1D dp)
- DP gives optimal solution but it's still pseudo-polynomial:
  - capacity W is a number, but input size is log(W)
  - Still NP-complete in theory

Unbounded knapsack - coin change (322/518)

- Goal: Min #coins to make exact amount (coins reusable)
- dp[i] = min # of coins to make amount i, len = amount + 1
- init: dp[0] = 0; all others = amount + 1 ("infinity")
- for each coin: for i := coin to amount:
  - dp[i] = min(dp[i], dp[i - coin] + 1)
- Final check:
- if dp[amount] == amount+1 → return -1 (not possible)
- else → return dp[amount] (result)
- Time: O(n * amount) | Space: O(amount)
- Ex: coins = [1, 2, 5]; amount = 13 → dp[13] = 3 (5 + 5 + 3)

Fractional Knapsack

- greedy optimal, not DP
- Sort items by value/weight ratio descending
- Loop through sorted items:
- if item fits: take whole item
- else: take fraction to fill remaining capacity
- remainingCapacity -= item.Weight (or part of it)
- totalValue += value taken (full or fraction)
- Time: O(n log n) for sorting | Space: O(1)

Longest Palindromic Subsequence (516)

- Given a string, find the length of the longest subsequence that is a palindrome
- A subsequence is not necessarily contiguous as substring
- A palindrome reads the same forward and backward
- dp[i][j] = length of LPS in substring s[i..j] (inclusive both)
- Base case (fill matrix first) → For i := range dp:
  - dp[i] = make([]int, n)
  - dp[i][i] = 1 (single char is a palindrome)
- Fill DP table bottom-up:
  - Outer loop → i := n - 1; i >= 0; i--
  - Inner loop → j = i + 1; j < n; j++
- In loop logic:
  - if characters match, extend the known palindrome within the bounds
  - else, take the maximum excluding either the left or the right character
- In loop code:
  - if s[i] == s[j] → dp[i][j] = dp[i+1][j-1] + 2
  - else → dp[i][j] = maxi(dp[i+1][j], dp[i][j-1])
- Final result: LPS len in the full string s[0..n-1] → dp[0][n-1]
- Time: O(n²), Space: O(n²)

House robber (198)

- Given an array of non-negative integers representing the amount of money at each house, return the maximum amount you can rob without robbing two adjacent houses
- dp[i] = max amount that can be robbed from house 0 to i
- For each house i, you decide to rob it or to skip it:
  - current = houses[i] + dp[i-2]
  - previousHouse = dp[i-1]
- dp[i] = maxi(currentHouse, previousHouse)
- Base cases:
  - dp[0] = houses[0]
  - dp[1] = maxi(houses[0], houses[1])
- Also acid: if n == 0 → return 0; if n == 1 → houses[1]
- Time: O(n); Space: O(n) → can be optimized to O(1)

Trie

Insert

```
- Add word to trie, reusing shared prefixes, only
- new branches are added, O(k), k = length of word
func Insert(word string) {
    node := t.Root
    for _, ch := range word {
        if node.Children[ch] == nil {
            // if child doesn't exist, create it
            node.Children[ch] = NewNode()
        }
        node = node.Children[ch]
        node.Count++ // Count = # of descendants
    }
    node.IsEnd = true // Mark last node as word end
}
```

Search

```
- Return true if word exists in trie
- Full match only, O(k), k = length of word
func Search(word string) bool {
    node := t.Root
    for _, ch := range word {
        if node.Children[ch] == nil {
            return false // no such char path
        }
        node = node.Children[ch]
    }
    return node.IsEnd // Must end on valid word
}
```

Starts with

```
- Return true if any word starts with prefix
- Partial match ok, O(k), k = prefix length
func StartsWith(prefix string) bool {
    node := t.Root
    for _, ch := range prefix {
        if node.Children[ch] == nil {
            return false // no such char path
        }
        node = node.Children[ch]
    }
    return true
}
```

Search with wildcard

```
- '.' matches any char, supports wildcard search
- DFS backtrack, worst case O(26^w)
func SearchWithWildcard(word string) bool {
    return dfs(t.Root, []rune(word), 0)
}

func dfs(n *Node, word []rune, i int) bool {
    if i == len(word) {
        return n.IsEnd
    }
    ch := word[i]
    if ch == '.' { // try every possible child
        for _, child := range n.Children {
            if dfs(child, word, i+1) {
                return true // early exit
            }
        }
        return false // none matched
    } // normal character case
    if n.Children[ch] == nil {
        return false // char is not in path
    }
    return dfs(n.Children[ch], word, i+1)
}
```

Prefix count

```
- returns # of words starting with prefix
- basically a variation of Autocomplete
func PrefixCount(prefix string) int {
    node := t.Root
    for _, ch := range prefix {
        if node.Children[ch] == nil {
            return 0 // no such prefix
        }
        node = node.Children[ch]
    } // node is at last prefix character
    count := 0
    countWords(node, &count)
    return count
}
// counts words under current node
func countWords(node *Node, count *int) {
    if node.IsEnd {
        *count++ // word ends here
    }
    for _, child := range node.Children {
        countWords(child, count)
    }
} // O(k + output), k = len(prefix)
```

Prefix count (fast/precomputed)

```
- returns # of words starting with prefix
- each node stores Count: number of words
- that pass through it (# of descendants)
func PrefixCountFast(prefix string) int {
    n := t.Root
    for _, ch := range prefix {
        if n.Children[ch] == nil {
            return 0
        }
        n = n.Children[ch]
    }
    return n.Count // precomputed on Insert
} // O(k), k = len(prefix)
```

Delete

- collect all nodes along the word path (root → last char) in a stack with entries (char rune, parent *Node)
- we need parent node reference because later we will need to delete the child node from its parent's Children map
- delete the word by unmarking IsEnd on the last node
- clean up in reverse (by popping from stack)
  - decrement count for each node
  - remove any child node if it's no longer needed (it has no children and it's not the end of any other word)
- O(k), where k = len(word)

Autocomplete

```
- Return all words that start with prefix
- O(k + output), DFS from last prefix char
func Autocomplete(prefix string) []string {
    node := t.Root
    for _, ch := range prefix {
        if node.Children[ch] == nil {
            return nil
        }
        node = node.Children[ch]
    } // node is at last prefix character
    var res []string
    collect(node, prefix, &res)
    return res
}
// perform DFS to collect all words
func collect(n *Node, path string, res *[]string) {
    if n.IsEnd {
        *res = append(*res, path)
    }
    for ch, child := range n.Children {
        collect(child, path+string(ch), res)
    }
}
```

Basic structures

```
type Node struct { 11 usages
    Children map[rune]*Node // Child nodes for ea
    IsEnd    bool           // Marks if this node
    Count    int
}
// Implement interface
type Trie struct { 15 usages
    Root *Node
}
func NewNode() *Node { 2 usages
    return &Node{Children: make(map[rune]*Node)}
}
func NewTrie() *Trie { 3 usages
    return &Trie{
        Root: NewNode(),
    }
}
func NewTrieFromArray(words ...string) *Trie { 6
    tr := NewTrie()
    for _, root := range words {
        tr.Insert(root)
    }
    return tr
}
```

Replace Words

```
- replaces words in the sentence with shortest matching root
- build trie - O(N*K), N: # of roots, K: avg root length
- replace words - O(M+L), M: # of words in sentence, L: avg word len
func ReplaceWords(dictionary []string, sentence string) string { 1 use
    tr := trie.NewTrieFromArray(dictionary...)
    words := strings.Split(sentence, sep: " ")
    for i, word := range words {// for each word in the sentence
        prefix := ""
        node := tr.Root              // start from root
        for _, ch := range word {// for each character in the word
            if node.Children[ch] == nil {
                break // no further prefix match
            }
            prefix += string(ch)     // update prefix
            node = node.Children[ch] // go to the next character
            if node.IsEnd {
                words[i] = prefix // replace word with shortest root
                break             // early exit
            }
        }
    }
    return strings.Join(words, sep: " ")
} // Total - O(N*K + M*L)
```

LeetCode problems

- LC 208 – Implement Trie (Prefix Tree): insert, search, startsWith
- LC 211– Design Add and Search Words: Trie + DFS for wildcard . matching
- LC 1804 – Implement Trie II (Prefix Count): adds word count and prefix count methods
- LC 648 – Replace Words: match and replace prefixes in a sentence using Trie
- LC 1268 – Search Suggestions System: autocomplete system, return top k prefix matches
- LC 1032 – Stream of Characters: suffix Trie, check if stream suffix matches any word
- LC 676 – Implement Magic Dictionary: DFS on Trie, allow one char difference
- LC 472 – Concatenated Words: Trie + DFS/Memo to check if word is made of other words
- LC 212 – Word Search II: DFS + Trie to search words in a 2D grid using prefix pruning

Backtrack DFS

Subset generation (78/90/131)

```
- get all subsets of length k, variations include:
vp to k or all. - just change/remove condition
func getSubsets(input []string, k int) [][]string {
    var result [][]string
    var dfs func(start int, path []string)
    dfs = func(start int, path []string) {
        if len(path) == k {
            pathCopy := append([]string{}, path...)
            result = append(result, pathCopy)
            return
        }
        for i := start; i < len(input); i++ {
            path = append(path, input[i])
            dfs(i+1, path)
            path = path[:len(path)-1] // backtrack
        }
    }
    dfs(start: 0, path: nil)
    return result
} // O(n*k), each level -> n branches, k levels
```

Combination sum (39/40/377/216)

- Given candidates and target, find all combinations that sum to target (reuse allowed)
- Use DFS with backtracking: choose → recurse → unchoose
- Only move forward: for i := start; to avoid duplicates like [2,3] vs [3,2]
- Allow reuse: recurse with same index i (not i+1)
- Track sum and path; stop early if sum > target
- Append deep copy of path when sum == target
- Same idea as subset generation but we accumulate sum
  - dfs(i, sum+num)
- Conceptually → coin change (unbounded knapsack), but here we collect all combinations, not just the length of smallest combination/set
- Time: O(2^target * k), where k = avg combination length
  - O(2^T) → exponential number of recursive paths
  - * k → each valid combination -> length up to k (copying costs O(k))
- Space: O(target / min + total output)
- Works well for small targets or when early pruning is used
- Backtracking preferred over DP when you need actual combinations, not just counts

Permutation generation (46/47)

```
// Generate permutations using i - base dfs
// `used` map prevents reusing the same element along the current
// recursive path, but it allows reusing previously used elements
func getPermutationsLengthK(input []string, k int) [][]string {
    var result [][]string
    var dfs func(used map[string]bool, path []string)
    dfs = func(used map[string]bool, path []string) {
        if len(path) == k {
            result = append(result, append([]string{}, path...))
            return
        }
        for i := 0; i < len(input); i++ {
            if used[input[i]] {
                continue // skip already used
            }
            used[input[i]] = true
            path = append(path, input[i]) // pick
            dfs(used, path)               // recurse
            path = path[:len(path)-1]     // unpick
            used[input[i]] = false        // reset flag
        }
    }
    dfs(make(map[string]bool), path: nil)
    return result
} // O(n / (n-k)!) -> generating permutations of k out of n
```

Heaps

Median from Data Stream (295/480)

- Use two heaps:
  - Max-heap left = lower half
  - Min-heap right = upper half
- Maintain: len(left) == len(right) or left = right + 1
- Invariant: max(left) ≤ median ≤ min(right)
- Get median:
  - Odd → left.Peek()
  - Even → (left.Peek() + right.Peek()) / 2
- AddNum:
  - If num ≤ left.Peek() → insert left, else insert right
  - Rebalance if size diff > 1
- Time (for each number) → O(n log n):
  - AddNum → O(log n)
  - GetMedian → O(1)
- Naive approach: sort each time → O(n log n)

Top K problems

K Smallest Elements (973/658/692/347)

- Use a max-heap of size k
- For each element in array:
- If heap has less than k elements → insert
- Else if current element < heap.Peek() → pop + insert
- Heap root always holds the largest of the smallest k
- O(n log k) for each el. we reheapify; Space: O(k)

K Largest Elements (215/703)

- Use a min-heap of size k
- For each element in array:
- If heap has less than k elements → insert
- Else if current element > heap.Peek() → pop + insert
- Heap root always holds the smallest of the largest k
- O(n log k) for each el. we reheapify; Space: O(k)

Tehniques

Power set generation

- Number of subsets: 2^n
- Use binary numbers to represent subsets
- ex. for set {A, B, C}, 101 is subset {A, C}
- Loop from 0 to 1<<n (2^n)
  - loop i...n
    - if s & (1 << i) != 0 → item in subset
- Time: O(n * 2^n) → exponential
- Practical for n ≤ 20

Generate all substrings

- generate all substrings starting from 0, 1...n-1
- for each, generate 1 len, 2 len, ...n len substrings
- code:
- for i := 0; i < len(str); i++ {
  - for j := i; j < len(str); j++ {
    - substr := str[i : j+1]

Reverse slice (manually)

```
func reverseSlice(arr []int) []int { 1 usage
    var reversed []int
    for i := range arr {
        reversed = append(reversed, arr[len(arr)-1-i])
    }
    return reversed
}
```

Binary Search

- Input array must be sorted
- left := 0; right := len(arr) - 1
- for: left <= right
  - mid = (left+right)/2
  - guess = arr[mid]
  - if guess == item → return mid
  - if guess was too high → right = mid - 1
  - if guess was too low → left = mid + 1
- return -1 → not found
- Halve each step: O(log n)

Sorting

Selection sort

- Repeatedly find min in unsorted part
- minIdx := i + findMin(arr[:i])
- arr[i], arr[minIdx] = arr[minIdx], arr[i]
- O(n^2)

Merge sort

- Divide & Conquer
- Recursively split; merge sorted halves
- Stable, consistent O(n log n)
- Space: O(n) (needs extra array)

Quicksort

- Pick pivot, partition into < and >
- Recursively sort left/right:
- [sorted left] + pivot + [sorted right]
- O(n log n), Worst: O(n^2) (bad pivot)