LECTURE 12 OF 12

# Advanced Topics: String Algorithms, Tries, and Dynamic Programming

**Data Structures & Algorithms**

Based on Robert Sedgewick & Kevin Wayne

*Algorithms, 4th Edition* (Addison-Wesley, 2011)

Chapters 5.1, 5.2, 5.3 and Supplementary Material

Duration: 4 hours (with breaks)

---

## Learning Objectives

1. Understand the structure and operations of R-way tries and ternary search tries (TSTs) for efficient string key storage, retrieval, prefix search, and wildcard matching.
2. Master the Knuth-Morris-Pratt (KMP) substring search algorithm, including the construction of the failure function (LPS array) and the deterministic finite automaton (DFA) approach presented by Sedgewick.
3. Learn the foundational principles of dynamic programming -- overlapping subproblems and optimal substructure -- and apply both top-down memoization and bottom-up tabulation strategies.
4. Solve classic dynamic programming problems including the Longest Common Subsequence (LCS) and the 0/1 Knapsack problem, including reconstruction of optimal solutions via backtracking.
5. Compare and contrast string search algorithms (brute force, KMP, Boyer-Moore, Rabin-Karp) and string symbol table implementations (tries, TSTs, hash tables, BSTs) in terms of time complexity, space usage, and practical applicability.

6. Synthesize the full breadth of data structures and algorithms covered across all twelve lectures into a coherent intellectual framework, identifying connections between topics and charting a path for continued study.

---

# Part I: Tries (Prefix Trees)

*Estimated time: 30 minutes*

## 1.1 Motivation: Why Tries?

Throughout this course we have studied general-purpose symbol tables backed by balanced BSTs (Lecture 5) and hash tables (Lecture 6). These structures treat keys as opaque objects compared via a total order or a hash function. When the keys are *strings* -- and strings are by far the most common key type in practice -- we can do substantially better by exploiting the *character-by-character structure* of the keys themselves.

Consider a symbol table containing N string keys drawn from an alphabet of size R. A balanced BST requires O(log N) key comparisons per search, but each comparison itself examines up to W characters (the length of the key), giving O(W log N) character accesses in the worst case. A hash table requires O(W) time to compute the hash plus O(1) expected time for the lookup, but hash tables do not support ordered operations such as finding all keys with a given prefix.

Tries -- the name comes from "re*trie*val," though it is conventionally pronounced "try" to distinguish it from "tree" -- offer a remarkable guarantee: search and insert operations examine at most W characters regardless of the number of keys N. Moreover, tries natively support prefix queries, longest-prefix matching, and wildcard matching, operations that are awkward or impossible with hash tables.

## 1.2 R-Way Tries: Definition and Structure

**DEFINITION.** An *R-way trie* is a tree-structured symbol table for string keys over an alphabet of size R. Each node contains:
- An optional *value* (non-null if the node marks the end of a key stored in the trie).
- An array of R *children*, one for each character in the alphabet. The child at index c corresponds to the character c.

The *root* represents the empty prefix. A key is stored by tracing a path from the root: the first character of the key selects a child of the root, the second character selects a child of that node, and so forth. The value is stored at the node reached after processing the last character.

```
Structure: TrieNode
    value        -- associated value, or NULL if this node is not a key endpoint
    children[0..R-1]  -- array of R child pointers, initially all NULL
```

For the standard lowercase English alphabet, R = 26. For extended ASCII, R = 256. For Unicode, R = 65536 -- which is why we will later discuss TSTs as a space-efficient alternative.

## 1.3 Trie Operations

### Insert

To insert a key-value pair (key, val) into the trie, we walk from the root, following the path dictated by successive characters of the key. If a child pointer along the path is NULL, we create a new node. When we reach the node corresponding to the last character, we store the value.

```
function TrieInsert(root, key, val):
    node = root
    for i = 0 to length(key) - 1:
        c = key[i]
        if node.children[c] is NULL:
            node.children[c] = new TrieNode()
        node = node.children[c]
    node.value = val
```

The time complexity is O(W) where W = length(key), independent of the number of keys in the trie.

### Search

To search for a key, we follow the same path. If at any point a child pointer is NULL, the key is not present. If we reach the final node but its value is NULL, the key is not present (the path exists only as a prefix of some longer key).

```
function TrieSearch(root, key):
    node = root
    for i = 0 to length(key) - 1:
        c = key[i]
        if node.children[c] is NULL:
            return NULL      -- key not found
        node = node.children[c]
    return node.value         -- may be NULL if key not stored
```

Time complexity: O(W) in all cases (hit or miss).

## Starts With (Prefix Search)

One of the most powerful features of tries is prefix search: given a prefix string, determine whether any key in the trie begins with that prefix, or collect all such keys.

```
function TrieStartsWith(root, prefix):
    node = root
    for i = 0 to length(prefix) - 1:
        c = prefix[i]
        if node.children[c] is NULL:
            return false
        node = node.children[c]
    return true   -- at least the prefix path exists
```

To collect all keys with a given prefix, we first navigate to the node representing the prefix, then perform a depth-first traversal collecting all keys below:

```
function TrieKeysWithPrefix(root, prefix):
    results = empty list
    node = root
    for i = 0 to length(prefix) - 1:
        c = prefix[i]
        if node.children[c] is NULL:
            return results   -- no keys with this prefix
        node = node.children[c]
    Collect(node, prefix, results)
    return results

function Collect(node, currentPrefix, results):
    if node is NULL:
        return
    if node.value is not NULL:
        results.append(currentPrefix)
    for c = 0 to R - 1:
        if node.children[c] is not NULL:
            Collect(node.children[c], currentPrefix + char(c), results)
```

This operation takes time proportional to the length of the prefix plus the number of keys collected, which is optimal.

## Longest Prefix Match

Given a query string, find the longest key in the trie that is a prefix of the query. This operation is fundamental in IP routing (longest-prefix match on network addresses) and in tokenization.

```
function TrieLongestPrefixOf(root, query):
    node = root
    longestLength = -1
    for i = 0 to length(query) - 1:
        if node.value is not NULL:
            longestLength = i
        c = query[i]
        if node.children[c] is NULL:
            break
        node = node.children[c]
    if node is not NULL and node.value is not NULL:
        longestLength = length(query)
    if longestLength == -1:
        return NULL
    return query[0..longestLength-1]
```

## Wildcard Matching

Tries can support pattern matching with wildcard characters (e.g., "." matches any single character). We use a recursive approach:

```
function TrieWildcardSearch(root, pattern):
    return WildcardHelper(root, pattern, 0)

function WildcardHelper(node, pattern, index):
    if node is NULL:
        return false
    if index == length(pattern):
        return node.value is not NULL
    c = pattern[index]
    if c == '.':
        for ch = 0 to R - 1:
            if WildcardHelper(node.children[ch], pattern, index + 1):
                return true
        return false
    else:
        return WildcardHelper(node.children[c], pattern, index + 1)
```

In the worst case (all wildcards), this examines every node in the trie, but for patterns with few wildcards, it is highly efficient.

## 1.4 Delete Operation

Deletion in a trie is more subtle than insertion. After removing the value from the target node, we must also clean up any nodes that have become unnecessary (no value and no children).

```
function TrieDelete(root, key):
    DeleteHelper(root, key, 0)

function DeleteHelper(node, key, depth):
    if node is NULL:
        return NULL
    if depth == length(key):
        node.value = NULL     -- remove the value
    else:
        c = key[depth]
        node.children[c] = DeleteHelper(node.children[c], key, depth + 1)

    -- If this node has a value, keep it
    if node.value is not NULL:
        return node

    -- If any child is non-null, keep this node
    for c = 0 to R - 1:
        if node.children[c] is not NULL:
            return node

    -- Node has no value and no children; delete it
    return NULL
```

## 1.5 Space Analysis

**PROPOSITION A.** The space consumed by an R-way trie containing N keys of average length W is between R_N (if keys share maximal prefixes) and R_N*W (if keys share no characters at all). Each node consumes O(R) space for the children array.

*Proof.* In the best case, all N keys share a common prefix of length W-1, requiring only N + W - 1 nodes, each with R child pointers. In the worst case, no prefixes are shared, requiring up to N * W nodes. Each node stores an array of R pointers. Thus total space ranges from O(R * N) to O(R * N * W). For typical English dictionary words with R = 256, this can be substantial: a trie with 100,000 keys of average length 8 could use up to 204 million pointers in the worst case. In practice, prefix sharing among natural language words significantly reduces the actual space used, but the worst-case bound remains a real concern for large alphabets.

## 1.6 Trie Trace Example

Let us trace the insertion of the keys "sea", "she", "shell", "shore", and "the" into an R-way trie (showing only relevant children).

```
Insert "sea":
    root -> [s] -> [e] -> [a]*

Insert "she":
    root -> [s] -> [e] -> [a]*
                -> [h] -> [e]*

Insert "shell":
    root -> [s] -> [e] -> [a]*
                -> [h] -> [e]* -> [l] -> [l]*

Insert "shore":
    root -> [s] -> [e] -> [a]*
                -> [h] -> [e]* -> [l] -> [l]*
                      -> [o] -> [r] -> [e]*

Insert "the":
    root -> [s] -> [e] -> [a]*
                -> [h] -> [e]* -> [l] -> [l]*
                      -> [o] -> [r] -> [e]*
         -> [t] -> [h] -> [e]*

    (* denotes a node with a non-null value)
```

Searching for "she" follows root -> s -> h -> e, finds a value: HIT.
Searching for "sho" follows root -> s -> h -> o, finds no value: MISS (but prefix path exists).
Searching for "see" follows root -> s -> e, but e has no child 'e': MISS.

## 1.7 Applications of Tries

Tries find application across a wide range of domains:

- **Autocomplete systems:** Type-ahead search in web browsers, IDEs, and search engines use tries to rapidly find all words sharing a common prefix with the user's partial input.
- **Spell checkers:** Tries combined with edit-distance algorithms allow efficient detection and correction of misspelled words.
- **IP routing:** Longest-prefix matching on binary representations of IP addresses uses binary tries (or compressed variants like Patricia tries) to route network packets.
- **T9 predictive text:** The old phone keypad text input system uses tries keyed on digit sequences to predict words.
- **Genome analysis:** DNA sequences over the alphabet {A, C, G, T} are stored in tries for rapid substring and prefix queries.

# Part II: Ternary Search Tries (TSTs)

*Estimated time: 20 minutes*

## 2.1 Motivation

As we noted, R-way tries have excellent time performance but potentially devastating space consumption for large alphabets. The ternary search trie (TST), introduced by Bentley and Sedgewick in 1997, is an elegant compromise that retains the time advantages of tries while dramatically reducing space usage.

The core idea is to replace the R-way branching at each node with a three-way branching that combines the character-by-character decomposition of tries with the comparison-based branching of binary search trees.

**DEFINITION.** A *ternary search trie (TST)* is a trie variant in which each node stores a single character and has exactly three children:

- **left**: points to a subtrie for characters *less than* the node's character.
- **mid** (or **equal**): points to a subtrie for the *next character* of keys whose current character matches the node's character.
- **right**: points to a subtrie for characters *greater than* the node's character.

```
Structure: TSTNode
    character   -- the character stored at this node
    value       -- associated value (NULL if not a key endpoint)
    left        -- subtrie for characters < this character
    mid         -- subtrie for next character when current character matches
    right       -- subtrie for characters > this character
```

## 2.2 TST Insert

```
function TSTInsert(node, key, val, depth):
    c = key[depth]
    if node is NULL:
        node = new TSTNode()
        node.character = c
    if c < node.character:
        node.left = TSTInsert(node.left, key, val, depth)
    else if c > node.character:
        node.right = TSTInsert(node.right, key, val, depth)
    else if depth < length(key) - 1:
        node.mid = TSTInsert(node.mid, key, val, depth + 1)
    else:
        node.value = val
    return node
```

When inserting a key, at each node we compare the current character of the key with the character stored at the node. If they match, we advance to the next character of the key and follow the mid pointer. If the key character is less, we go left; if greater, we go right. This is exactly analogous to how a BST would handle character comparisons, but applied one character position at a time.

## 2.3 TST Search

```
function TSTSearch(node, key, depth):
    if node is NULL:
        return NULL
    c = key[depth]
    if c < node.character:
        return TSTSearch(node.left, key, depth)
    else if c > node.character:
        return TSTSearch(node.right, key, depth)
    else if depth < length(key) - 1:
        return TSTSearch(node.mid, key, depth + 1)
    else:
        return node.value
```

## 2.4 TST Prefix Search and Collection

TSTs support prefix operations analogously to R-way tries. To find all keys with a given prefix, we first search for the prefix (navigating to the node where the last character of the prefix is matched), then collect all keys in the subtrie rooted at that node's mid child:

```
function TSTKeysWithPrefix(root, prefix):
    results = empty list
    node = root
    for depth = 0 to length(prefix) - 1:
        c = prefix[depth]
        while node is not NULL:
            if c < node.character:
                node = node.left
            else if c > node.character:
                node = node.right
            else:
                break  -- character matched
        if node is NULL:
            return results  -- prefix not in trie
        if depth < length(prefix) - 1:
            node = node.mid
    -- node now points to the node matching the last char of prefix
    if node.value is not NULL:
        results.append(prefix)
    TSTCollect(node.mid, prefix, results)
    return results

function TSTCollect(node, currentPrefix, results):
    if node is NULL:
        return
    TSTCollect(node.left, currentPrefix, results)
    if node.value is not NULL:
        results.append(currentPrefix + node.character)
    TSTCollect(node.mid, currentPrefix + node.character, results)
    TSTCollect(node.right, currentPrefix, results)
```

## 2.5 Space and Time Analysis

**PROPOSITION B.** A TST containing N string keys of average length W uses O(N * W) space, with each node consuming O(1) space (three pointers plus a character plus a value field). This is a factor of R improvement over the R-way trie.

*Proof.* Each character in each key corresponds to at most one node in the TST (the node where the character is matched via the mid pointer). Additional nodes may be created along left/right chains, but the total number of nodes is bounded by N * W * constant. Each node stores a fixed number of fields (character, value, three pointers), so the total space is O(N * W). Compared to the R-way trie where each node stores R pointers, the TST reduces the per-node cost from O(R) to O(1), yielding an overall factor-of-R space improvement.

**PROPOSITION C.** Search in a TST examines at most W + log N characters, where W is the key length and N is the number of keys.

*Proof sketch.* At each node, we either advance to the next character of the key (via the mid pointer, which can happen at most W times) or move left/right without advancing (which constitutes a BST search among characters at this position, bounded by the height of the local BST). If the TST is reasonably balanced (or if keys are inserted in random order), the left/right searches contribute O(log N) comparisons in total. Therefore, the total number of node accesses is O(W + log N).

## 2.6 Comparison: R-Way Tries vs. TSTs vs. Hash Tables

| Property | R-Way Trie | TST | Hash Table |
|---|---|---|---|
| Search (hit) | O(W) | O(W + log N) | O(W) expected |
| Search (miss) | O(log_R N) typical | O(log N) | O(W) expected |
| Insert | O(W) | O(W + log N) | O(W) expected |
| Space per node | O(R) | O(1) | -- |
| Total space | O(R * N * W) worst | O(N * W) | O(N * W) |
| Prefix search | Excellent (native) | Good (native) | Not supported |
| Longest prefix | Yes | Yes | Not supported |
| Ordered iteration | Yes | Yes | No |
| Wildcard search | Yes | Yes (slower) | No |
| Implementation | Simple | Moderate | Simple |

The TST is often the best choice in practice for string symbol tables when ordered operations or prefix queries are needed, as it combines near-trie time performance with near-BST space efficiency. Hash tables remain the best choice when only exact-match lookups are needed, ordering is unnecessary, and the key set does not demand prefix-based queries.

## 2.7 TST Trace Example

Insert "sea", "she", "shell" into a TST:

```
Insert "sea":
         s
         |mid
         e
         |mid
         a*

Insert "she" (s matches, h > e so go right of e):
         s
         |mid
         e ---right--- h
         |mid          |mid
         a*            e*

Insert "shell" (s matches, h found via right of e, e matches, then l, l):
         s
         |mid
         e ---right--- h
         |mid          |mid
         a*            e*
                       |mid
                       l
                       |mid
                       l*
```

Searching for "she": s(match, go mid) -> e(h > e, go right) -> h(match, go mid) -> e(match, depth == len-1, return value) -> HIT.

Searching for "shell": s(match, mid) -> e(h > e, right) -> h(match, mid) -> e(match, mid) -> l(match, mid) -> l(match, depth == len-1, return value) -> HIT.

Searching for "see": s(match, mid) -> e(e == e, match, mid) -> a(e > a, go right) -> NULL -> MISS.

---

# Part III: KMP String Matching

*Estimated time: 35 minutes*

## 3.1 The Substring Search Problem

The *substring search problem* (also called *string matching* or *pattern matching*) is one of the most fundamental problems in computer science:

> ***Given:*** *A text string T of length N and a pattern string P of length M.*
> ***Find:*** *The first occurrence of P as a contiguous substring of T (or report that P does not occur in T).*

This problem arises constantly in text editors (find and replace), search engines (query matching), bioinformatics (DNA and protein sequence matching), network intrusion detection (packet payload scanning), and countless other applications. The efficiency of substring search has direct practical impact on the performance of many widely-used systems.

## 3.2 Brute-Force Approach

The naive approach slides the pattern across the text one position at a time, checking for a complete match at each position:

```
function BruteForceSearch(text, pattern):
    N = length(text)
    M = length(pattern)
    for i = 0 to N - M:
        j = 0
        while j < M and text[i + j] == pattern[j]:
            j = j + 1
        if j == M:
            return i        -- match found at position i
    return -1               -- no match
```

**PROPOSITION D.** The brute-force substring search algorithm has worst-case time complexity O(N * M).

*Proof.* At each of the N - M + 1 starting positions, we may compare up to M characters before discovering a mismatch. This occurs, for example, when searching for the pattern "AAAB" in the text "AAAAAAAAB" -- at each starting position, we compare A characters until reaching the B, discovering the mismatch only at the M-th character. Total comparisons: (N - M + 1) * M = O(N * M). In the specific example, searching for "AAAB" (M=4) in "AAAAAAAAB" (N=9) requires (9-4+1)*4 = 24 comparisons in the worst case.

In practice, for natural language text over a large alphabet, the brute-force approach works tolerably well because mismatches tend to occur after very few character comparisons (the probability of matching even two characters at a random position is roughly 1/R^2 for alphabet size R). However, the quadratic worst case is a real concern in applications such as DNA sequence matching (where the alphabet is small: {A, C, G, T}) and in adversarial settings (e.g., denial-of-service attacks exploiting worst-case behavior in network packet inspection).

## 3.3 KMP: The Key Insight

The Knuth-Morris-Pratt (KMP) algorithm, published in 1977 by Donald Knuth, James Morris, and Vaughan Pratt, achieves O(N + M) worst-case time by exploiting a crucial insight:

> ***When a mismatch occurs after matching several characters, the pattern itself contains enough information to determine where the next potential match could begin, without re-examining characters of the text that have already been matched.***

In the brute-force approach, when we discover a mismatch at position j in the pattern (meaning we have already matched pattern[0..j-1] with some substring of the text), we back up in the text to position i+1 and restart the comparison from pattern[0]. KMP *never* backs up in the text. Instead, it uses precomputed information about the internal structure of the pattern to determine how far to shift the pattern before resuming comparisons.

The key observation is: at the point of mismatch, we know that text[i-j..i-1] = pattern[0..j-1] (these characters matched). If there is a proper prefix of pattern[0..j-1] that is also a suffix of pattern[0..j-1], then that prefix is already aligned with the text. We can shift the pattern so that this prefix becomes the new partial match and continue from where we left off in the text.

## 3.4 The Failure Function (LPS Array)

The *failure function* (also called the *Longest Proper Prefix which is also Suffix* array, or LPS array) is the heart of KMP. For each position j in the pattern, lps[j] is the length of the longest proper prefix of pattern[0..j] that is also a suffix of pattern[0..j].

**DEFINITION.** For a pattern P of length M, the *LPS array* is defined as:
- lps[0] = 0 (by convention; a single character has no proper prefix that is also a suffix).
- For j >= 1: lps[j] = length of the longest string that is both a proper prefix of P[0..j] and a suffix of P[0..j].

A "proper" prefix of a string S is any prefix of S that is not S itself. For example, the proper prefixes of "ABAB" are "", "A", "AB", "ABA".

**Example:** For pattern "ABABAC":
- lps[0] = 0 (single char "A")
- lps[1] = 0 ("AB" has no proper prefix that is also a suffix)
- lps[2] = 1 ("ABA" -- prefix "A" = suffix "A")
- lps[3] = 2 ("ABAB" -- prefix "AB" = suffix "AB")
- lps[4] = 3 ("ABABA" -- prefix "ABA" = suffix "ABA")
- lps[5] = 0 ("ABABAC" -- no proper prefix equals a suffix)

The key property is: when a mismatch occurs at pattern position j after matching j characters, the LPS value lps[j-1] tells us the length of the longest prefix of the pattern that is already matched at the current position in the text. We can therefore shift the pattern to align this prefix and continue comparing from position lps[j-1] in the pattern.

## 3.5 Computing the LPS Array

The LPS array itself can be computed in O(M) time using a bootstrap technique -- the same KMP logic applied to the pattern matching against itself:

```
function BuildLPSArray(pattern):
    M = length(pattern)
    lps = new array of size M
    lps[0] = 0
    len = 0        -- length of previous longest prefix-suffix
    i = 1
    while i < M:
        if pattern[i] == pattern[len]:
            len = len + 1
            lps[i] = len
            i = i + 1
        else:
            if len != 0:
                len = lps[len - 1]
                -- do NOT increment i; try a shorter prefix-suffix
            else:
                lps[i] = 0
                i = i + 1
    return lps
```

The algorithm maintains a variable `len` representing the length of the current candidate prefix-suffix. When characters match, we extend the prefix-suffix and record it. When they do not match, we fall back to the next-longest prefix-suffix (given by lps[len-1]) and try again. This fallback is the same principle used in the search itself.

**PROPOSITION E.** The LPS array computation runs in O(M) time.

*Proof.* We maintain two quantities: the index i (which ranges from 1 to M-1) and the variable `len`. The index i increases by 1 in the first and third branches of the if-else. In the second branch (the fallback), `len` strictly decreases (since lps[len-1] < len). The total number of times `len` can decrease is bounded by the total number of times it has increased, which is at most M-1 (since it increases by 1 each time and starts at 0). Therefore the total number of iterations of the while loop is at most (M-1) + (M-1) = 2M - 2 = O(M).

## 3.6 KMP Search Algorithm

With the LPS array precomputed, the KMP search proceeds as follows:

```
function KMPSearch(text, pattern):
    N = length(text)
    M = length(pattern)
    if M == 0:
        return 0
    lps = BuildLPSArray(pattern)
    i = 0          -- index into text
    j = 0          -- index into pattern
    while i < N:
        if text[i] == pattern[j]:
            i = i + 1
            j = j + 1
        if j == M:
            return i - j   -- match found at position i - j
            -- To find ALL matches: record i - j, then set j = lps[j - 1]
        else if i < N and text[i] != pattern[j]:
            if j != 0:
                j = lps[j - 1]
            else:
                i = i + 1
    return -1       -- no match found
```

**PROPOSITION F.** The KMP search algorithm runs in O(N) time (after O(M) preprocessing), for a total of O(N + M).

*Proof.* Consider the quantity 2i - j. At each iteration of the while loop, this quantity increases by at least 1: in the match case, i increases by 1 and j increases by 1, so 2i - j increases by 2 - 1 = 1. In the mismatch case with j != 0, j decreases (so -j increases), and i stays the same, so 2i - j increases. In the mismatch case with j == 0, i increases by 1. Since 2i - j starts at 0 and is bounded by 2N (since i <= N and j >= 0), the total number of iterations is at most 2N = O(N). Combined with O(M) preprocessing, the total is O(N + M).

## 3.7 KMP Trace Example

Let us trace KMP on:

- Text: `A B A B A B A C A B`
- Pattern: `A B A B A C`

**Step 1: Build LPS for "ABABAC"**

```
Pattern: A  B  A  B  A  C
Index:   0  1  2  3  4  5

i=1, len=0: P[1]='B' vs P[0]='A' -> mismatch, len=0 -> lps[1]=0

i=2, len=0: P[2]='A' vs P[0]='A' -> match, len=1 -> lps[2]=1

i=3, len=1: P[3]='B' vs P[1]='B' -> match, len=2 -> lps[3]=2

i=4, len=2: P[4]='A' vs P[2]='A' -> match, len=3 -> lps[4]=3

i=5, len=3: P[5]='C' vs P[3]='B' -> mismatch
     len = lps[2] = 1
     P[5]='C' vs P[1]='B' -> mismatch
     len = lps[0] = 0
     P[5]='C' vs P[0]='A' -> mismatch, len=0
     lps[5] = 0

LPS array: [0, 0, 1, 2, 3, 0]
```

**Step 2: Search**

```
Text:    A B A B A B A C A B
Index:   0 1 2 3 4 5 6 7 8 9

i=0, j=0: T[0]='A' == P[0]='A' -> i=1, j=1
i=1, j=1: T[1]='B' == P[1]='B' -> i=2, j=2
i=2, j=2: T[2]='A' == P[2]='A' -> i=3, j=3
i=3, j=3: T[3]='B' == P[3]='B' -> i=4, j=4
i=4, j=4: T[4]='A' == P[4]='A' -> i=5, j=5
i=5, j=5: T[5]='B' != P[5]='C' -> MISMATCH!
           j != 0, so j = lps[4] = 3
           (We know T[2..4] = "ABA" = P[0..2], so we can continue from j=3)

i=5, j=3: T[5]='B' == P[3]='B' -> i=6, j=4
i=6, j=4: T[6]='A' == P[4]='A' -> i=7, j=5
i=7, j=5: T[7]='C' == P[5]='C' -> i=8, j=6
j=6 == M=6 -> MATCH FOUND at position i - j = 8 - 6 = 2
```

Pattern "ABABAC" found at index 2 in the text. Notice that the text pointer i *never* moved backwards. When the mismatch occurred at i=5, j=5, instead of backing up to text position 1, KMP consulted the LPS array: lps[4] = 3 tells us that the first 3 characters of the pattern ("ABA") are already aligned with text positions 2, 3, 4. So we shift the pattern and continue comparing P[3] with T[5].

Total character comparisons: 11 (vs. up to 30 for brute force in the worst case for these lengths).

## 3.8 Sedgewick's DFA Approach

Sedgewick and Wayne present an alternative (and equivalent) formulation of KMP using an explicit *deterministic finite automaton (DFA)*. Rather than storing the LPS array and using conditional logic during the search, the DFA approach precomputes a complete transition table that maps every (state, character) pair to the next state.

The DFA has M+1 states (0 through M), where state j means "the last j characters of the text examined so far match the first j characters of the pattern." State M is the accept state. The DFA transition table dfa[c][j] gives the next state when the machine is in state j and reads character c from the text.

```
function BuildDFA(pattern, R):
    M = length(pattern)
    dfa = new 2D array [R][M], initialized to 0
    dfa[pattern[0]][0] = 1
    X = 0               -- restart state (state to simulate mismatch transitions)
    for j = 1 to M - 1:
        for c = 0 to R - 1:
            dfa[c][j] = dfa[c][X]       -- copy mismatch transitions from restart state
        dfa[pattern[j]][j] = j + 1      -- override: match transition advances state
        X = dfa[pattern[j]][X]          -- update restart state
    return dfa

function DFASearch(text, pattern, R):
    N = length(text)
    M = length(pattern)
    dfa = BuildDFA(pattern, R)
    j = 0
    for i = 0 to N - 1:
        j = dfa[text[i]][j]
        if j == M:
            return i - M + 1    -- match found
    return -1
```

The DFA approach has several notable properties:

1. **The search loop is branchless** (aside from the match check): a single array lookup per character. This makes it extremely fast in practice and suitable for hardware implementations.
2. **No backup:** Like the LPS version, the DFA never re-examines a text character.
3. **Space:** The DFA table has size R * M. For large alphabets, this can be significant (e.g., R = 256, M = 100 requires 25,600 entries). The LPS approach uses only O(M) space.
4. **Construction time:** O(R * M) for the DFA vs. O(M) for the LPS array.

The restart state X is the key to the DFA construction. It represents the state the DFA would be in if we had fed it the pattern substring pattern[1..j-1] (the pattern shifted right by one). This is exactly the information needed to handle mismatches: if we are in state j and read a character c != pattern[j], we should transition to the same state we would reach from state X reading character c.

## 3.9 DFA Trace for "ABABAC" (R=3: A=0, B=1, C=2)

```
j=0: dfa[A][0]=1, X=0
j=1: copy from X=0: dfa[A][1]=dfa[A][0]=1, dfa[B][1]=dfa[B][0]=0, dfa[C][1]=dfa[C][0]=0
     override: dfa[B][1]=2
     update X: X=dfa[B][0]=0
j=2: copy from X=0: dfa[A][2]=1, dfa[B][2]=0, dfa[C][2]=0
     override: dfa[A][2]=3
     update X: X=dfa[A][0]=1
j=3: copy from X=1: dfa[A][3]=1, dfa[B][3]=2, dfa[C][3]=0
     override: dfa[B][3]=4
     update X: X=dfa[B][1]=2
j=4: copy from X=2: dfa[A][4]=3, dfa[B][4]=0, dfa[C][4]=0
     override: dfa[A][4]=5
     update X: X=dfa[A][2]=3
j=5: copy from X=3: dfa[A][5]=1, dfa[B][5]=4, dfa[C][5]=0
     override: dfa[C][5]=6
     (X would update but we stop)

DFA Table:
       state0 state1 state2 state3 state4 state5
  A:     1      1      3      1      5      1
  B:     0      2      0      4      0      4
  C:     0      0      0      0      0      6
```

## 3.10 Comparison with Other String Search Algorithms

| Algorithm | Preprocessing | Search (worst) | Search (typical) | Backs up text? | Extra space |
|---|---|---|---|---|---|
| Brute force | O(1) | O(NM) | O(N) | Yes | O(1) |
| KMP (LPS) | O(M) | O(N) | O(N) | No | O(M) |
| KMP (DFA) | O(RM) | O(N) | O(N) | No | O(RM) |
| Boyer-Moore | O(M + R) | O(NM)* | O(N/M) | Yes | O(R + M) |
| Rabin-Karp | O(M) | O(NM) | O(N + M) | No | O(1) |

*Boyer-Moore achieves O(N) worst case with the good-suffix rule; O(NM) with bad-character rule alone.

**Boyer-Moore** (briefly): Scans the pattern right-to-left and uses two heuristics (bad-character and good-suffix) to skip large portions of the text. When a mismatch occurs, the bad-character rule shifts the pattern to align the mismatched text character with its rightmost occurrence in the pattern. On average, it examines only about N/M characters, making it sublinear and the fastest algorithm in practice for large alphabets and long patterns. It is the algorithm used by `grep` and most text editors.

**Rabin-Karp** (briefly): Uses a rolling hash function to compute the hash of each M-character substring of the text in O(1) amortized time (using modular arithmetic). If the hash matches the pattern's hash, a full character-by-character comparison verifies the match. Expected O(N + M) time; O(NM) worst case due to hash collisions. The primary advantage of Rabin-Karp is its easy generalization to multi-pattern search and 2D pattern matching.

---

# Part IV: Dynamic Programming Introduction

*Estimated time: 25 minutes*

## 4.1 What Is Dynamic Programming?

Dynamic programming (DP) is an algorithm design technique that solves complex optimization and counting problems by decomposing them into simpler *overlapping subproblems*, solving each subproblem only once, and storing (caching) the results for future reference. The term was coined by Richard Bellman in the 1950s while working at the RAND Corporation.

DP is arguably the most versatile algorithm design technique after divide-and-conquer and greedy algorithms. It applies to an enormous range of problems: shortest paths (Bellman-Ford, which we studied in Lecture 9, is a DP algorithm), sequence alignment, parsing, resource allocation, scheduling, and countless others.

DP applies when a problem has two key properties:

**DEFINITION.** A problem exhibits *optimal substructure* if an optimal solution to the problem can be constructed efficiently from optimal solutions to its subproblems. In other words, the principle of optimality holds: any sub-policy of an optimal policy is itself optimal.

**DEFINITION.** A problem has *overlapping subproblems* if the space of subproblems is "small" -- that is, a recursive algorithm for the problem solves the same subproblems over and over rather than always generating new subproblems. This is in contrast to divide-and-conquer (e.g., mergesort), where the subproblems are disjoint.

When both properties hold, we can avoid redundant computation by storing the results of solved subproblems in a table (an array, matrix, or hash map) and looking them up in O(1) time instead of recomputing them. This typically transforms an exponential-time recursive algorithm into a polynomial-time algorithm -- often a dramatic improvement.

## 4.2 Two Approaches: Memoization vs. Tabulation

There are two standard approaches to implementing a DP solution:

**Top-Down with Memoization.** Write the natural recursive solution, but augment it with a lookup table (the "memo"). Before computing the answer to a subproblem, check if it has already been computed and stored in the memo. If so, return the stored result immediately. Otherwise, compute it recursively, store the result in the memo, and then return it.

Advantages of memoization:
- Often easier to write: just add caching to the natural recursion.
- Computes only the subproblems that are actually needed (lazy evaluation).
- The recurrence structure is explicit in the code.

Disadvantages:
- Recursion overhead (function call stack).
- For deep recursion, may cause stack overflow.
- Hash table lookups may be slower than array indexing.

**Bottom-Up with Tabulation.** Determine the topological order in which subproblems depend on each other (smallest/simplest subproblems first), then iteratively fill in a table starting from the base cases and building up to the desired answer.

Advantages of tabulation:
- No recursion overhead.
- Better cache performance (sequential memory access patterns).
- Easier to optimize space (e.g., keeping only the last row or two of a 2D table).

Disadvantages:
- Must determine the evaluation order explicitly.
- May compute subproblems that are not needed for the final answer.
- The recurrence relationship may be less obvious in the code.

Both approaches compute the same answers and have the same asymptotic time complexity. The choice between them is largely a matter of taste and practical considerations.

## 4.3 The Canonical Example: Fibonacci Numbers

The Fibonacci sequence is defined by: F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2) for n >= 2.

This is the standard introductory example for DP because the naive recursive solution has dramatic overlapping subproblems, and the DP solution offers an equally dramatic speedup.

### Naive Recursion

```
function FibNaive(n):
    if n <= 1:
        return n
    return FibNaive(n - 1) + FibNaive(n - 2)
```

This has exponential time complexity. The recursion tree for FibNaive(5) reveals massive redundancy:

```
                         F(5)
                        /    \
                     F(4)     F(3)
                    /   \     /   \
                 F(3)  F(2) F(2)  F(1)
                /  \   / \   / \
             F(2) F(1) F(1) F(0) F(1) F(0)
             / \
          F(1) F(0)
```

F(3) is computed 2 times, F(2) is computed 3 times, F(1) is computed 5 times, F(0) is computed 3 times. In total, 15 function calls to compute F(5) = 5.

**PROPOSITION G.** The naive recursive Fibonacci algorithm has time complexity O(phi^n) where phi = (1 + sqrt(5)) / 2 ~ 1.618 is the golden ratio.

*Proof sketch.* Let T(n) be the number of additions performed by FibNaive(n). Then T(0) = T(1) = 0 and T(n) = T(n-1) + T(n-2) + 1 for n >= 2. This recurrence is dominated by the Fibonacci recurrence itself: T(n) >= F(n) for all n >= 1. Since F(n) = (phi^n - psi^n) / sqrt(5) where psi = (1 - sqrt(5))/2, and |psi| < 1, we have F(n) = Theta(phi^n). A matching upper bound can be shown similarly, giving T(n) = Theta(phi^n). For practical purposes: F(50) requires over 10^10 additions; F(100) requires over 10^20. The naive algorithm is hopelessly slow for even moderate n.

## Memoized (Top-Down) DP

```
function FibMemo(n, memo):
    if n <= 1:
        return n
    if memo[n] is defined:
        return memo[n]
    memo[n] = FibMemo(n - 1, memo) + FibMemo(n - 2, memo)
    return memo[n]
```

Each value F(k) for k = 0, 1, ..., n is computed exactly once. Total additions: n - 1. Time: O(n). Space: O(n) for the memo table plus O(n) for the recursion call stack.

## Tabulated (Bottom-Up) DP

```
function FibTable(n):
    if n <= 1:
        return n
    dp = new array of size n + 1
    dp[0] = 0
    dp[1] = 1
    for i = 2 to n:
        dp[i] = dp[i - 1] + dp[i - 2]
    return dp[n]
```

Time: O(n). Space: O(n). This can be further optimized to O(1) space since we only ever need the two most recent values:

```
function FibOptimized(n):
    if n <= 1:
        return n
    prev2 = 0
    prev1 = 1
    for i = 2 to n:
        current = prev1 + prev2
        prev2 = prev1
        prev1 = current
    return prev1
```

The progression from O(phi^n) to O(n) to O(n) with O(1) space illustrates the power of recognizing and eliminating redundant computation.

## 4.4 The Systematic DP Approach

For any DP problem, follow this four-step methodology:

1. **Define subproblems.** What does the table entry dp[i] (or dp[i][j], etc.) represent? State this clearly and precisely. The choice of subproblem definition is the most creative and important step.
2. **Write the recurrence.** Express dp[i] in terms of smaller (previously solved) subproblems. This is where the mathematical insight lies. The recurrence must capture the optimal substructure of the problem.
3. **Identify base cases.** What are the values of dp[...] for the simplest, smallest subproblems? These are the starting points from which the entire table is built.
4. **Determine evaluation order.** In what order must the table be filled so that when we compute dp[i], all subproblems it depends on have already been solved? For 1D problems this is typically left-to-right; for 2D problems, it is typically row-by-row from top-left to bottom-right.

An optional fifth step is:

1. **Reconstruct the solution.** The DP table gives the *value* of the optimal solution. To find the solution itself (which items to select, which characters to include, etc.), backtrack through the table to determine which choices were made at each step.

This systematic approach transforms DP from a mysterious art into a structured engineering methodology. The hard part is step 1 (choosing the right subproblem definition) and step 2 (finding the recurrence). Steps 3-5 are largely mechanical once the recurrence is established.

---

# Part V: Longest Common Subsequence (LCS)

*Estimated time: 25 minutes*

## 5.1 Problem Definition

**DEFINITION.** A *subsequence* of a string S is a sequence of characters obtained from S by deleting zero or more characters without changing the relative order of the remaining characters. For example, "ACE" is a subsequence of "ABCDE" (delete B and D), as are "ABCDE" itself and the empty string "".

Note the critical distinction: a *substring* must be contiguous characters; a *subsequence* need not be contiguous. Every substring is a subsequence, but not every subsequence is a substring.

**DEFINITION.** The *Longest Common Subsequence (LCS)* of two strings S1 and S2 is the longest string that is a subsequence of both S1 and S2. The LCS is not necessarily unique -- there may be multiple common subsequences of the same maximum length.

For example:
- S1 = "ABCBDAB"
- S2 = "BDCABA"
- One LCS = "BCBA" (length 4)
- Another LCS = "BDAB" (length 4)

The LCS problem has widespread applications:
- **Bioinformatics:** Comparing DNA or protein sequences to identify evolutionary relationships.
- **Version control:** The `diff` utility uses LCS to compute the differences between two file versions.
- **Plagiarism detection:** Measuring the similarity of two documents.
- **Data compression:** LCS-based techniques can identify repeated patterns.

## 5.2 Why Not Brute Force?

A string of length N has 2^N subsequences (each character is either included or excluded). Comparing all subsequences of S1 against all subsequences of S2 would take O(2^N * 2^M) time -- completely infeasible for strings of any significant length.

## 5.3 Optimal Substructure

Let S1 have length N and S2 have length M. Define the subproblem as finding the LCS of prefixes S1[0..i-1] and S2[0..j-1]. Consider the last characters:

- **If S1[i-1] == S2[j-1]:** This common character can always be appended to the LCS of the shorter prefixes. So LCS(S1[0..i-1], S2[0..j-1]) ends with this character, and its length is 1 + LCS_length(S1[0..i-2], S2[0..j-2]).
- **If S1[i-1] != S2[j-1]:** At least one of these characters does not participate in the LCS. We take the better of two options: either the last character of S1 is not in the LCS (solve LCS(S1[0..i-2], S2[0..j-1])) or the last character of S2 is not in the LCS (solve LCS(S1[0..i-1], S2[0..j-2])).

This exhibits optimal substructure. The subproblems overlap extensively: the recursive tree for LCS has exponential size without memoization, but only O(N*M) distinct subproblems.

## 5.4 DP Recurrence

Define dp[i][j] = length of the LCS of S1[0..i-1] and S2[0..j-1].

**Base cases:** dp[0][j] = 0 for all j (empty S1 has no common subsequence with anything). dp[i][0] = 0 for all i (empty S2).

**Recurrence:**

```
if S1[i-1] == S2[j-1]:
    dp[i][j] = dp[i-1][j-1] + 1
else:
    dp[i][j] = max(dp[i-1][j], dp[i][j-1])
```

The answer is dp[N][M].

## 5.5 Full Pseudocode

```
function LCS(S1, S2):
    N = length(S1)
    M = length(S2)
    dp = new 2D array of size (N+1) x (M+1), initialized to 0

    for i = 1 to N:
        for j = 1 to M:
            if S1[i-1] == S2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    return dp[N][M]
```

## 5.6 Backtracking to Reconstruct the LCS

The DP table gives us the *length* of the LCS. To find the actual subsequence, we backtrack from dp[N][M] to dp[0][0], tracing the decisions that led to each cell's value:

```
function ReconstructLCS(dp, S1, S2):
    i = length(S1)
    j = length(S2)
    result = empty string
    while i > 0 and j > 0:
        if S1[i-1] == S2[j-1]:
            result = S1[i-1] + result    -- prepend this character
            i = i - 1
            j = j - 1
        else if dp[i-1][j] >= dp[i][j-1]:
            i = i - 1                    -- came from above
        else:
            j = j - 1                    -- came from the left
    return result
```

The backtracking takes O(N + M) time (at each step, either i or j decreases by 1).

## 5.7 Complete DP Table Trace

Let S1 = "ABCB" (N=4) and S2 = "BDCAB" (M=5).

**The DP Table:**

```
         ""  B  D  C  A  B
    ""    0  0  0  0  0  0
     A    0  0  0  0  1  1
     B    0  1  1  1  1  2
     C    0  1  1  2  2  2
     B    0  1  1  2  2  3
```

**Detailed filling (row by row):**

Row i=1 (S1[0]='A'):
- j=1: 'A' != 'B' -> max(dp[0][1], dp[1][0]) = max(0, 0) = 0
- j=2: 'A' != 'D' -> max(dp[0][2], dp[1][1]) = max(0, 0) = 0
- j=3: 'A' != 'C' -> max(dp[0][3], dp[1][2]) = max(0, 0) = 0

- j=4: 'A' == 'A' -> dp[0][3] + 1 = 0 + 1 = 1
- j=5: 'A' != 'B' -> max(dp[0][5], dp[1][4]) = max(0, 1) = 1

Row i=2 (S1[1]='B'):
- j=1: 'B' == 'B' -> dp[1][0] + 1 = 0 + 1 = 1
- j=2: 'B' != 'D' -> max(dp[1][2], dp[2][1]) = max(0, 1) = 1
- j=3: 'B' != 'C' -> max(dp[1][3], dp[2][2]) = max(0, 1) = 1
- j=4: 'B' != 'A' -> max(dp[1][4], dp[2][3]) = max(1, 1) = 1
- j=5: 'B' == 'B' -> dp[1][4] + 1 = 1 + 1 = 2

Row i=3 (S1[2]='C'):
- j=1: 'C' != 'B' -> max(dp[2][1], dp[3][0]) = max(1, 0) = 1
- j=2: 'C' != 'D' -> max(dp[2][2], dp[3][1]) = max(1, 1) = 1
- j=3: 'C' == 'C' -> dp[2][2] + 1 = 1 + 1 = 2
- j=4: 'C' != 'A' -> max(dp[2][4], dp[3][3]) = max(1, 2) = 2
- j=5: 'C' != 'B' -> max(dp[2][5], dp[3][4]) = max(2, 2) = 2

Row i=4 (S1[3]='B'):
- j=1: 'B' == 'B' -> dp[3][0] + 1 = 0 + 1 = 1
- j=2: 'B' != 'D' -> max(dp[3][2], dp[4][1]) = max(1, 1) = 1
- j=3: 'B' != 'C' -> max(dp[3][3], dp[4][2]) = max(2, 1) = 2
- j=4: 'B' != 'A' -> max(dp[3][4], dp[4][3]) = max(2, 2) = 2
- j=5: 'B' == 'B' -> dp[3][4] + 1 = 2 + 1 = 3

**LCS length = dp[4][5] = 3.**

**Backtracking to reconstruct:**
- i=4, j=5: S1[3]='B' == S2[4]='B' -> take 'B', move to (3, 4)
- i=3, j=4: S1[2]='C' != S2[3]='A', dp[2][4]=1 < dp[3][3]=2 -> go left to (3, 3)
- i=3, j=3: S1[2]='C' == S2[2]='C' -> take 'C', move to (2, 2)
- i=2, j=2: S1[1]='B' != S2[1]='D', dp[1][2]=0 < dp[2][1]=1 -> go left to (2, 1)
- i=2, j=1: S1[1]='B' == S2[0]='B' -> take 'B', move to (1, 0)
- i=1, j=0: j=0, stop.

**LCS = "BCB"** (length 3). Reading the taken characters in order: B, C, B.

## 5.8 Complexity Analysis

**PROPOSITION H.** The LCS algorithm runs in O(N * M) time and O(N * M) space.

*Proof.* The DP table has (N+1) * (M+1) entries. Each entry is computed in O(1) time from at most three previously computed entries (dp[i-1][j-1], dp[i-1][j], and dp[i][j-1]). The total computation is therefore O(N * M). The backtracking to reconstruct the LCS takes O(N + M) additional time. The total space for the DP table is O(N * M).

**Space Optimization.** If we only need the *length* of the LCS (not the actual subsequence), we can reduce space to O(min(N, M)) by observing that computing row i of the DP table requires only row i-1. We maintain two rows and alternate between them:

```
function LCS_SpaceOptimized(S1, S2):
    -- Ensure S2 is the shorter string for minimal space
    if length(S1) < length(S2):
        swap S1 and S2
    N = length(S1)
    M = length(S2)
    prev = new array of size M+1, initialized to 0
    curr = new array of size M+1, initialized to 0

    for i = 1 to N:
        for j = 1 to M:
            if S1[i-1] == S2[j-1]:
                curr[j] = prev[j-1] + 1
            else:
                curr[j] = max(prev[j], curr[j-1])
        swap prev and curr
        fill curr with 0

    return prev[M]
```

This uses O(min(N, M)) space while maintaining O(N * M) time. Reconstructing the actual LCS from only two rows is possible but requires more sophisticated techniques (e.g., Hirschberg's algorithm, which combines the space-efficient forward computation with a backward computation using divide-and-conquer to achieve O(min(N,M)) space with full LCS reconstruction).

# Part VI: 0/1 Knapsack Problem

*Estimated time: 20 minutes*

## 6.1 Problem Definition

**DEFINITION.** The *0/1 Knapsack Problem*: Given N items, each with a positive integer weight w_i and a positive integer value v_i, and a knapsack with integer capacity W, find the subset of items that maximizes the total value subject to the constraint that the total weight does not exceed W. Each item is either fully included (1) or fully excluded (0) -- there is no fractional inclusion (which would be the *fractional knapsack*, solvable by a simple greedy algorithm).

Formally: maximize the sum of v_i * x_i over all items i, subject to the sum of w_i * x_i <= W, where x_i is in {0, 1}.

The 0/1 knapsack problem is one of the most fundamental problems in combinatorial optimization. It models resource allocation, capital budgeting, cargo loading, project selection, and many other practical scenarios. It is also one of Karp's 21 NP-complete problems, meaning no polynomial-time algorithm is known (in the input size, which is O(N + log W)).

## 6.2 Optimal Substructure

Consider item i (the i-th item in our enumeration). There are two cases:
- **Item i is NOT included in the optimal solution:** The optimal value comes from selecting the best subset of items 1..i-1 with the full capacity W.
- **Item i IS included in the optimal solution:** The optimal value is v_i plus the best value achievable from items 1..i-1 with reduced capacity W - w_i.

We take the better of the two options (or only the first option if w_i > W, since item i is too heavy to include).

## 6.3 DP Recurrence

Define dp[i][w] = maximum value achievable using only items 1..i with a knapsack of capacity w.

**Base cases:** dp[0][w] = 0 for all w >= 0 (no items means no value). dp[i][0] = 0 for all i >= 0 (zero capacity means nothing can be included).

**Recurrence:**

```
if w_i > w:
    dp[i][w] = dp[i-1][w]                                -- item i too heavy; skip it
else:
    dp[i][w] = max(dp[i-1][w], dp[i-1][w - w_i] + v_i)  -- max of excluding vs. including item i
```

The answer is dp[N][W].

## 6.4 Full Pseudocode

```
function Knapsack01(weights, values, N, W):
    dp = new 2D array of size (N+1) x (W+1), initialized to 0

    for i = 1 to N:
        for w = 1 to W:
            if weights[i-1] > w:
                dp[i][w] = dp[i-1][w]
            else:
                dp[i][w] = max(dp[i-1][w], dp[i-1][w - weights[i-1]] + values[i-1])

    return dp[N][W]
```

## 6.5 Backtracking to Find Selected Items

The DP table gives us the maximum achievable value. To determine *which* items to include, we backtrack from dp[N][W]:

```
function KnapsackItems(dp, weights, values, N, W):
    selected = empty list
    w = W
    for i = N down to 1:
        if dp[i][w] != dp[i-1][w]:
            -- Item i was included (its inclusion changed the optimal value)
            selected.prepend(i)
            w = w - weights[i-1]
    return selected
```

The logic is straightforward: if dp[i][w] equals dp[i-1][w], then item i was not needed for the optimal solution at this capacity. If they differ, item i must have been included (its inclusion improved the value), so we record it and reduce the remaining capacity by its weight.

## 6.6 Complete Trace Example

Consider 4 items and capacity W = 7:

| Item | Weight | Value |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 3 | 4 |
| 3 | 4 | 5 |
| 4 | 5 | 7 |

**DP Table:**

```
       w=0  w=1  w=2  w=3  w=4  w=5  w=6  w=7
i=0     0    0    0    0    0    0    0    0
i=1     0    1    1    1    1    1    1    1
i=2     0    1    1    4    5    5    5    5
i=3     0    1    1    4    5    6    6    9
i=4     0    1    1    4    5    7    8    9
```

**Filling step by step:**

Row i=1 (item 1: weight=1, value=1):
- w=1: weight 1 <= 1 -> max(dp[0][1], dp[0][0]+1) = max(0, 1) = 1
- w=2: weight 1 <= 2 -> max(dp[0][2], dp[0][1]+1) = max(0, 1) = 1
- w=3: weight 1 <= 3 -> max(dp[0][3], dp[0][2]+1) = max(0, 1) = 1
- w=4 through w=7: same pattern, all = 1 (only item 1 available, value 1)

Row i=2 (item 2: weight=3, value=4):
- w=1: weight 3 > 1 -> dp[1][1] = 1
- w=2: weight 3 > 2 -> dp[1][2] = 1
- w=3: weight 3 <= 3 -> max(dp[1][3], dp[1][0]+4) = max(1, 4) = 4
- w=4: weight 3 <= 4 -> max(dp[1][4], dp[1][1]+4) = max(1, 5) = 5
- w=5: weight 3 <= 5 -> max(dp[1][5], dp[1][2]+4) = max(1, 5) = 5
- w=6: weight 3 <= 6 -> max(dp[1][6], dp[1][3]+4) = max(1, 5) = 5
- w=7: weight 3 <= 7 -> max(dp[1][7], dp[1][4]+4) = max(1, 5) = 5

Row i=3 (item 3: weight=4, value=5):
- w=1,2,3: weight 4 > w -> copy from row 2: 1, 1, 4
- w=4: weight 4 <= 4 -> max(dp[2][4], dp[2][0]+5) = max(5, 5) = 5
- w=5: weight 4 <= 5 -> max(dp[2][5], dp[2][1]+5) = max(5, 6) = 6
- w=6: weight 4 <= 6 -> max(dp[2][6], dp[2][2]+5) = max(5, 6) = 6
- w=7: weight 4 <= 7 -> max(dp[2][7], dp[2][3]+5) = max(5, 9) = 9

Row i=4 (item 4: weight=5, value=7):
- w=1,2,3,4: weight 5 > w -> copy from row 3: 1, 1, 4, 5
- w=5: weight 5 <= 5 -> max(dp[3][5], dp[3][0]+7) = max(6, 7) = 7
- w=6: weight 5 <= 6 -> max(dp[3][6], dp[3][1]+7) = max(6, 8) = 8
- w=7: weight 5 <= 7 -> max(dp[3][7], dp[3][2]+7) = max(9, 8) = 9

**Maximum value = dp[4][7] = 9.**

**Backtracking to find selected items:**
- i=4, w=7: dp[4][7]=9 == dp[3][7]=9 -> item 4 NOT included
- i=3, w=7: dp[3][7]=9 != dp[2][7]=5 -> item 3 IS included, w = 7 - 4 = 3
- i=2, w=3: dp[2][3]=4 != dp[1][3]=1 -> item 2 IS included, w = 3 - 3 = 0
- i=1, w=0: dp[1][0]=0 == dp[0][0]=0 -> item 1 NOT included

**Selected items: {2, 3}. Total weight = 3 + 4 = 7. Total value = 4 + 5 = 9.**

Observe that items 2 and 3 perfectly fill the knapsack capacity (total weight = 7 = W) and achieve a total value of 9, which is better than any other feasible combination. For instance, items {1, 4} would give weight 6 and value 8, and items {1, 2, 3} would give weight 8 > W = 7 (infeasible).

## 6.7 Complexity Analysis

**PROPOSITION I.** The 0/1 Knapsack DP algorithm runs in O(N * W) time and O(N * W) space.

*Proof.* The DP table has (N+1) * (W+1) entries. Each entry is computed in O(1) time by examining at most two previously computed entries (dp[i-1][w] and dp[i-1][w-w_i]). The backtracking takes O(N) time. Total time: O(N * W). Total space: O(N * W).

**Pseudo-polynomial complexity.** The running time O(N * W) is *pseudo-polynomial*: it is polynomial in N and the *numeric value* of W, but not polynomial in the *size of the input*. The input size is O(N * (log V + log W)) where V is the largest value and log W is the number of bits needed to represent W. If W is encoded in binary, a W written with b bits can represent values up to 2^b, making the running time O(N * 2^b) -- exponential in the input size. This is why the 0/1 Knapsack problem remains NP-complete despite having this DP solution.

**Space optimization.** Since row i depends only on row i-1, we can reduce space to O(W) by keeping only two rows. When iterating, we must be careful to process w from W down to w_i (not from w_i up to W) to avoid using updated values from the current row:

```
function Knapsack01_SpaceOptimized(weights, values, N, W):
    dp = new array of size W+1, initialized to 0
    for i = 1 to N:
        for w = W down to weights[i-1]:
            dp[w] = max(dp[w], dp[w - weights[i-1]] + values[i-1])
    return dp[W]
```

This elegant 1D formulation uses O(W) space. The reverse iteration over w ensures that dp[w - weights[i-1]] still holds the value from the previous row (item i-1 and below).

# Part VII: Course Retrospective

*Estimated time: 15 minutes*

## 7.1 The Full Map: Twelve Lectures of Algorithms

We have reached the conclusion of a journey through the essential data structures and algorithms that form the foundation of computer science. Let us step back and survey the landscape we have traversed.

**Lecture 1: Introduction and Analysis of Algorithms.** We established the mathematical framework for analyzing algorithms: asymptotic notation (Big-O, Big-Omega, Big-Theta), the RAM model of computation, and techniques for counting primitive operations. This lecture provided the common language we have used throughout the course to discuss efficiency, compare algorithms, and make informed engineering decisions.

**Lecture 2: Stacks, Queues, and Elementary Data Structures.** We studied the fundamental abstract data types -- stacks (LIFO), queues (FIFO), deques, and their implementations via resizing arrays and linked lists. We introduced the amortized analysis of dynamic arrays. These structures appear as building blocks in virtually every algorithm we studied subsequently: DFS uses a stack, BFS uses a queue, Dijkstra's algorithm uses a priority queue.

**Lecture 3: Elementary Sorting (Selection Sort, Insertion Sort, Shellsort).** We began our study of sorting with the simplest O(N^2) algorithms. Despite their asymptotic inferiority, these algorithms taught us fundamental concepts: inversions, stability, adaptivity, and in-place computation. Insertion sort remains the algorithm of choice for small arrays and nearly-sorted inputs. Shellsort, with its ingenious diminishing-increment strategy, bridges the gap to the efficient algorithms.

**Lecture 4: Efficient Sorting (Mergesort, Quicksort).** We studied the two great divide-and-conquer sorting algorithms. Mergesort guarantees O(N log N) worst-case performance through stable merging. Quicksort achieves O(N log N) expected time through randomized partitioning and is the fastest general-purpose sorting algorithm in practice. We proved the Omega(N log N) lower bound for comparison-based sorting, establishing that mergesort and quicksort are asymptotically optimal.

**Lecture 5: Binary Search Trees and Balanced Trees.** We introduced the BST as the canonical ordered symbol table implementation. Unbalanced BSTs can degrade to O(N) height (equivalent to a linked list), motivating self-balancing trees: 2-3 trees and their elegant red-black tree encoding, which guarantee O(log N) worst-case height and therefore O(log N) search, insert, and delete.

**Lecture 6: Hash Tables.** We studied the unordered symbol table via hashing: hash function design (division, multiplication, universal hashing), collision resolution (separate chaining, linear probing, double hashing), dynamic resizing with load factor management, and the expected O(1) cost of search and insert. We compared hash tables with BSTs and identified the fundamental tradeoff: hash tables are faster for point queries but do not support ordered operations.

**Lecture 7: Priority Queues and Heapsort.** We studied the binary heap -- a complete binary tree stored in an array -- as the efficient implementation of the priority queue ADT. We learned the swim (percolate up) and sink (percolate down) operations, and the elegant in-place heapsort algorithm: the only comparison sort that is simultaneously in-place and O(N log N) worst-case.

**Lecture 8: Graph Fundamentals (Representations, BFS, DFS).** We entered the world of graphs, which arguably represent the single most versatile modeling tool in computer science. We studied adjacency list and adjacency matrix representations, depth-first search (with its applications to cycle detection, topological sort, and connected components) and breadth-first search (shortest paths in unweighted graphs).

**Lecture 9: Weighted Graphs and Shortest Paths (Dijkstra, Bellman-Ford).** We extended graph algorithms to weighted edges. Dijkstra's algorithm finds single-source shortest paths in O(E log V) time for graphs with non-negative edge weights. Bellman-Ford handles arbitrary (including negative) edge weights in O(VE) time and can detect negative-weight cycles. Both algorithms exemplify the "relaxation" technique for shortest-path computation.

**Lecture 10: Minimum Spanning Trees (Kruskal, Prim).** We studied the MST problem and two greedy algorithms grounded in the cut property. Kruskal's algorithm processes edges in weight order, adding each edge that does not create a cycle (using Union-Find for cycle detection). Prim's algorithm grows the MST from a single vertex, always adding the cheapest edge crossing the cut between the tree and the remaining vertices (using a priority queue). Both achieve O(E log V) time.

**Lecture 11: Union-Find.** We studied the disjoint-set data structure for dynamic connectivity. Starting from the naive quick-find (O(1) find, O(N) union) and quick-union (O(N) worst case for both), we developed weighted union (O(log N)) and path compression. The combination of weighted union and path compression achieves nearly O(1) amortized time per operation -- specifically O(alpha(N)), where alpha is the inverse Ackermann function that grows so slowly it is effectively constant for all practical purposes.

**Lecture 12: Advanced Topics (This Lecture).** We have concluded with string algorithms (tries and TSTs for string symbol tables, KMP for substring search) and dynamic programming (the systematic approach to optimization over overlapping subproblems, illustrated by Fibonacci, LCS, and 0/1 Knapsack). These topics broaden our algorithmic toolkit into pattern matching, string processing, and combinatorial optimization.

## 7.2 Connecting the Themes

Several grand themes weave through all twelve lectures:

**Abstraction and ADTs.** We consistently separated the *interface* of a data structure (what operations it supports -- push/pop for stacks, insert/deleteMin for priority queues, put/get for symbol tables) from the *implementation* (how those operations are realized using arrays, linked lists, trees, or hash tables). This separation is the intellectual foundation of software engineering and allows us to swap implementations without changing client code.

**The Time-Space Tradeoff.** Every structure involves a tradeoff. Hash tables use extra space for fast expected-time lookups. Tries use space proportional to R per node for O(W) time. The DFA version of KMP uses O(RM) space for the simplest possible inner loop. Understanding and navigating these tradeoffs is the essence of algorithmic engineering.

**Divide and Conquer.** Mergesort, quicksort, BSTs, and the KMP LPS construction all break problems into smaller pieces, solve them recursively, and combine results. This paradigm naturally leads to O(N log N) time for sorting and O(log N) time for searching.

**Greedy Algorithms.** Dijkstra, Prim, and Kruskal make locally optimal choices that provably lead to globally optimal solutions. The correctness of greedy algorithms always rests on a structural argument: the cut property for MSTs, the relaxation invariant for shortest paths.

**Dynamic Programming.** DP, the newest paradigm in our toolkit, applies when greedy fails and brute force is too expensive. By caching solutions to overlapping subproblems, DP achieves polynomial time for problems that would otherwise require exponential exploration. Many problems that seem to require enumerating all possibilities yield to DP.

**Graph Thinking.** An extraordinary range of real-world problems can be modeled as graph problems. Social networks, web link structures, dependency chains, road networks, scheduling constraints, circuit layouts -- all are graphs. Mastering BFS, DFS, Dijkstra, Bellman-Ford, Kruskal, and Prim equips you to tackle a vast family of problems.

## 7.3 What to Study Next

This course has covered the essential core of algorithms and data structures. Here are productive directions for continued study:

**Advanced Data Structures:** B-trees and B+ trees (database indexing), Fibonacci heaps (improving Dijkstra and Prim to O(E + V log V)), skip lists (probabilistic balanced search), splay trees (self-adjusting BSTs with amortized O(log N)), van Emde Boas trees (O(log log U) predecessor queries), suffix arrays and suffix trees (advanced string processing), segment trees and Fenwick trees (range queries and updates).

**Advanced Graph Algorithms:** Network flow (Ford-Fulkerson, Edmonds-Karp, push-relabel), bipartite matching (Hungarian algorithm, Hopcroft-Karp), strongly connected components (Tarjan's and Kosaraju's algorithms), all-pairs shortest paths (Floyd-Warshall, Johnson's algorithm), Euler and Hamilton paths, planarity testing.

**Advanced Dynamic Programming:** DP on trees, bitmask DP, digit DP, interval DP, profile DP. DP optimizations: divide-and-conquer optimization, Knuth's optimization, convex hull trick, the SMAWK algorithm. These techniques appear frequently in competitive programming and algorithm research.

**NP-Completeness and Intractability:** Understanding the P vs. NP question, NP-completeness reductions, and what it means for a problem to be "hard." This theoretical framework helps you recognize when a problem likely has no efficient exact solution, directing you toward approximation algorithms, heuristics, or restricted problem instances.

**Approximation and Randomized Algorithms:** Approximation algorithms for NP-hard problems (vertex cover, set cover, TSP), randomized algorithms (randomized quickselect, Miller-Rabin primality, random sampling), probabilistic data structures (Bloom filters, Count-Min sketch, HyperLogLog, locality-sensitive hashing).

**Competitive Programming:** An excellent way to develop algorithmic intuition and speed. Platforms such as Codeforces, LeetCode, AtCoder, and Project Euler provide thousands of problems at every difficulty level. Regular practice builds the pattern-recognition skills needed to quickly identify the right technique for a new problem.

## 7.4 The Big-Picture Message

The study of algorithms is ultimately about disciplined problem-solving. Each algorithm we have studied represents a distillation of human insight: the observation that sorting by divide-and-conquer yields O(N log N); that balanced trees guarantee logarithmic operations; that hashing converts comparison-based search into expected constant time; that memoizing overlapping subproblems converts exponential recursion into polynomial iteration.

These insights were contributed by some of the most brilliant minds in the history of computer science and mathematics: Knuth, Dijkstra, Kruskal, Prim, Bellman, Sedgewick, Hoare, Rabin, Karp, Morris, Pratt, Tarjan, and many others. Each algorithm is a small masterpiece of logical reasoning, and the collective body of algorithmic knowledge is one of the great intellectual achievements of the twentieth century.

As you continue your studies and careers, the specific algorithms you have learned will serve you well. But more importantly, the *way of thinking* that algorithm design cultivates -- precise problem definition, structural analysis, correctness reasoning, efficiency analysis, and iterative refinement -- will serve you in every technical endeavor.

---

# Part VIII: Practice Problems

## Problem 1: Trie Insert and Search Trace

**Problem:** Insert the keys "cat", "car", "card", "care", "cats" into an empty R-way trie, then trace the search for "card" and "can".

**Solution:**

After inserting all five keys (showing only the relevant branch structure):

```
root
 |
[c]
 |
[a]
 |
[t]* ("cat")------ [r]* ("car")
 |                  |
[s]* ("cats")      [d]* ("card") ---- [e]* ("care")
```

More precisely, from the root:
- root.children['c'] -> node_c
- node_c.children['a'] -> node_a
- node_a.children['t'] -> node_t (value = "cat")

- node_t.children['s'] -> node_s (value = "cats")
- node_a.children['r'] -> node_r (value = "car")
- node_r.children['d'] -> node_d (value = "card")
- node_r.children['e'] -> node_e (value = "care")

**Search for "card":**
1. root -> children['c'] = node_c (exists)
2. node_c -> children['a'] = node_a (exists)
3. node_a -> children['r'] = node_r (exists)
4. node_r -> children['d'] = node_d (exists)
5. node_d.value is not NULL -> HIT, return value associated with "card"

**Search for "can":**
1. root -> children['c'] = node_c (exists)
2. node_c -> children['a'] = node_a (exists)
3. node_a -> children['n'] = NULL -> MISS, return NULL

The search for "can" terminates at node_a because there is no child for 'n'. Only 3 character comparisons are needed to determine that "can" is not in the trie.

## Problem 2: KMP Failure Function (LPS) Computation

**Problem:** Compute the LPS array for the pattern "AABAABAAA".

**Solution:**

```
Pattern: A  A  B  A  A  B  A  A  A
Index:   0  1  2  3  4  5  6  7  8

lps[0] = 0 (by definition)

i=1, len=0:
  P[1]='A' == P[0]='A' -> match
  len = 1, lps[1] = 1

i=2, len=1:
  P[2]='B' != P[1]='A' -> mismatch
  len != 0, so len = lps[0] = 0
  P[2]='B' != P[0]='A' -> mismatch
  len == 0, so lps[2] = 0

i=3, len=0:
  P[3]='A' == P[0]='A' -> match
  len = 1, lps[3] = 1

i=4, len=1:
  P[4]='A' == P[1]='A' -> match
  len = 2, lps[4] = 2

i=5, len=2:
  P[5]='B' == P[2]='B' -> match
  len = 3, lps[5] = 3

i=6, len=3:
  P[6]='A' == P[3]='A' -> match
  len = 4, lps[6] = 4

i=7, len=4:
  P[7]='A' == P[4]='A' -> match
  len = 5, lps[7] = 5

i=8, len=5:
  P[8]='A' != P[5]='B' -> mismatch
  len != 0, so len = lps[4] = 2
  P[8]='A' != P[2]='B' -> mismatch
```

```
   len != 0, so len = lps[1] = 1
   P[8]='A' == P[1]='A' -> match
   len = 2, lps[8] = 2

Final LPS array: [0, 1, 0, 1, 2, 3, 4, 5, 2]
```

**Verification of lps[7]=5:** P[0..7] = "AABAABAAA". Nope, let us check: P[0..7] = "AABAABAA". The prefix of length 5 is "AABAA" and the suffix of length 5 is "AABAA". These are equal. The prefix of length 6 would be "AABAAB" and the suffix of length 6 is "BAABAA". These are not equal. So lps[7] = 5 is correct.

**Verification of lps[8]=2:** P[0..8] = "AABAABAAA". The prefix of length 2 is "AA" and the suffix of length 2 is "AA". These are equal. The prefix of length 3 is "AAB" and the suffix of length 3 is "AAA". These are NOT equal. So lps[8] = 2 is correct.

---

## Problem 3: LCS DP Table Fill

**Problem:** Find the LCS of S1 = "AGCAT" and S2 = "GAC". Show the complete DP table and reconstruct the LCS.

**Solution:**

```
        ""  G  A  C
   ""    0  0  0  0
    A    0  0  1  1
    G    0  1  1  1
    C    0  1  1  2
    A    0  1  2  2
    T    0  1  2  2
```

**Detailed filling:**

Row i=1 (S1[0]='A'):
- j=1: 'A' != 'G' -> max(dp[0][1], dp[1][0]) = max(0, 0) = 0
- j=2: 'A' == 'A' -> dp[0][1] + 1 = 0 + 1 = 1
- j=3: 'A' != 'C' -> max(dp[0][3], dp[1][2]) = max(0, 1) = 1

Row i=2 (S1[1]='G'):
- j=1: 'G' == 'G' -> dp[1][0] + 1 = 0 + 1 = 1
- j=2: 'G' != 'A' -> max(dp[1][2], dp[2][1]) = max(1, 1) = 1
- j=3: 'G' != 'C' -> max(dp[1][3], dp[2][2]) = max(1, 1) = 1

Row i=3 (S1[2]='C'):
- j=1: 'C' != 'G' -> max(dp[2][1], dp[3][0]) = max(1, 0) = 1
- j=2: 'C' != 'A' -> max(dp[2][2], dp[3][1]) = max(1, 1) = 1
- j=3: 'C' == 'C' -> dp[2][2] + 1 = 1 + 1 = 2

Row i=4 (S1[3]='A'):
- j=1: 'A' != 'G' -> max(dp[3][1], dp[4][0]) = max(1, 0) = 1
- j=2: 'A' == 'A' -> dp[3][1] + 1 = 1 + 1 = 2
- j=3: 'A' != 'C' -> max(dp[3][3], dp[4][2]) = max(2, 2) = 2

Row i=5 (S1[4]='T'):
- j=1: 'T' != 'G' -> max(dp[4][1], dp[5][0]) = max(1, 0) = 1
- j=2: 'T' != 'A' -> max(dp[4][2], dp[5][1]) = max(2, 1) = 2
- j=3: 'T' != 'C' -> max(dp[4][3], dp[5][2]) = max(2, 2) = 2

**LCS length = dp[5][3] = 2.**

**Backtracking:**
- i=5, j=3: 'T' != 'C'. dp[4][3]=2 >= dp[5][2]=2 -> go up to (4, 3)
- i=4, j=3: 'A' != 'C'. dp[3][3]=2 >= dp[4][2]=2 -> go up to (3, 3)
- i=3, j=3: 'C' == 'C' -> take 'C', move to (2, 2)
- i=2, j=2: 'G' != 'A'. dp[1][2]=1 >= dp[2][1]=1 -> go up to (1, 2)
- i=1, j=2: 'A' == 'A' -> take 'A', move to (0, 1)
- i=0: stop.

**LCS = "AC"** (length 2).

---

## Problem 4: 0/1 Knapsack Trace

**Problem:** Solve the 0/1 knapsack problem with capacity W = 5 and items:

| Item | Weight | Value |
|---|---|---|
| 1 | 2 | 3 |
| 2 | 3 | 4 |
| 3 | 4 | 5 |
| 4 | 5 | 6 |

**Solution:**

```
       w=0  w=1  w=2  w=3  w=4  w=5
i=0     0    0    0    0    0    0
i=1     0    0    3    3    3    3
i=2     0    0    3    4    4    7
i=3     0    0    3    4    5    7
i=4     0    0    3    4    5    7
```

**Row i=1** (weight=2, value=3):
- w=1: 2 > 1 -> dp[0][1] = 0
- w=2: 2 <= 2 -> max(dp[0][2], dp[0][0]+3) = max(0, 3) = 3
- w=3: 2 <= 3 -> max(dp[0][3], dp[0][1]+3) = max(0, 3) = 3
- w=4: 2 <= 4 -> max(dp[0][4], dp[0][2]+3) = max(0, 3) = 3
- w=5: 2 <= 5 -> max(dp[0][5], dp[0][3]+3) = max(0, 3) = 3

**Row i=2** (weight=3, value=4):
- w=1: 3 > 1 -> 0
- w=2: 3 > 2 -> 3
- w=3: 3 <= 3 -> max(dp[1][3], dp[1][0]+4) = max(3, 4) = 4
- w=4: 3 <= 4 -> max(dp[1][4], dp[1][1]+4) = max(3, 4) = 4
- w=5: 3 <= 5 -> max(dp[1][5], dp[1][2]+4) = max(3, 7) = 7

**Row i=3** (weight=4, value=5):
- w=1,2,3: 4 > w -> copy from row 2: 0, 3, 4
- w=4: 4 <= 4 -> max(dp[2][4], dp[2][0]+5) = max(4, 5) = 5
- w=5: 4 <= 5 -> max(dp[2][5], dp[2][1]+5) = max(7, 5) = 7

**Row i=4** (weight=5, value=6):
- w=1,2,3,4: 5 > w -> copy from row 3: 0, 3, 4, 5
- w=5: 5 <= 5 -> max(dp[3][5], dp[3][0]+6) = max(7, 6) = 7

**Maximum value = 7.**

**Backtracking:**
- i=4, w=5: dp[4][5]=7 == dp[3][5]=7 -> item 4 NOT included
- i=3, w=5: dp[3][5]=7 == dp[2][5]=7 -> item 3 NOT included
- i=2, w=5: dp[2][5]=7 != dp[1][5]=3 -> item 2 IS included. w = 5 - 3 = 2
- i=1, w=2: dp[1][2]=3 != dp[0][2]=0 -> item 1 IS included. w = 2 - 2 = 0

**Selected items: {1, 2}. Total weight = 2 + 3 = 5. Total value = 3 + 4 = 7.**

---

## Problem 5: Compare String Search Algorithms

**Problem:** For each scenario, recommend the best string search algorithm and justify your choice.

**(a)** Searching for a short pattern (5 characters) in a large English text (1 million characters), single query.

**Answer: Boyer-Moore.** For natural language text with a large alphabet (R = 26 or more for English), Boyer-Moore's bad-character heuristic allows it to skip large portions of the text, achieving sublinear average-case performance of approximately N/M character examinations. For a 5-character pattern in 1 million characters of English, Boyer-Moore would examine roughly 200,000 characters on average -- about 5x fewer than the text length. KMP would examine all N characters (though each only once). Brute force would also average about N comparisons for random English text, but Boyer-Moore's skipping gives it a clear advantage.

**(b)** Searching for a pattern in a binary string (alphabet = {0, 1}).

**Answer: KMP.** With a binary alphabet, Boyer-Moore's bad-character skip distance is limited to at most 1 position (since every character appears frequently in both pattern and text). This eliminates Boyer-Moore's primary advantage. KMP's guaranteed O(N + M) worst case and its efficiency with small alphabets make it the clear winner. The DFA variant is especially attractive here: with R = 2, the DFA table has only 2 * M entries, using very little space.

**(c)** Searching for 100 different patterns in the same text.

**Answer: Rabin-Karp** (multi-pattern variant) or the **Aho-Corasick** algorithm (a generalization of KMP to multiple patterns, not covered in this course). Standard Rabin-Karp can compute rolling hashes for all patterns simultaneously, achieving O(N * k) expected time where k is the number of patterns, which is better than running KMP or Boyer-Moore 100 separate times. Aho-Corasick achieves O(N + M_total + z) where M_total is the total pattern length and z is the number of matches. For 100 patterns, running any single-pattern algorithm 100 times gives 100 * N character examinations; Aho-Corasick reduces this to a single pass over the text.

**(d)** Streaming text that can only be read once (no backup allowed).

**Answer: KMP (DFA variant).** KMP never backs up the text pointer, making it the ideal choice for streaming data where rewinding is impossible (network packets, pipe input, tape storage). The DFA formulation is especially clean: one table lookup per input character, no conditional branches in the inner loop, constant time per character regardless of match or mismatch. Boyer-Moore's right-to-left scanning requires backup, and brute force requires backup on mismatch, making neither suitable for true streaming.

---

## Problem 6: TST vs. R-Way Trie vs. Hash Table

**Problem:** You need to store 50,000 English words (average length 8) and support: (a) exact lookup, (b) autocomplete (prefix search), (c) spell-check suggestions (find all words within edit distance 1). Which data structure would you choose and why?

**Solution:**

**(a) Exact lookup:**
- Hash table: O(W) expected (hash computation + O(1) amortized lookup). Very fast.
- R-way trie: O(W) worst case. Also very fast.
- TST: O(W + log N) typical. Slightly slower but still efficient.
- All three handle exact lookup well.

**(b) Autocomplete (prefix search):**
- Hash table: Cannot efficiently find all keys with a given prefix. Would require scanning all 50,000 entries -- O(N * W) time. Unacceptable for interactive autocomplete.
- R-way trie: Native support. Navigate to the prefix node in O(P) time (P = prefix length), then collect all descendants. Extremely efficient.
- TST: Also supports prefix search natively, with similar efficiency to the R-way trie (slightly more node

traversals due to left/right branching).
- Hash tables are eliminated by this requirement.

**(c) Spell-check (edit distance 1):**
- TST/Trie: Can be adapted for fuzzy matching by exploring neighboring branches during recursive traversal, allowing one insertion, deletion, or substitution at each level. The trie structure naturally constrains the search space.
- Hash table: Would require generating all possible strings at edit distance 1 from the query word. For a word of length W over alphabet size R, there are approximately R * W insertions + W deletions + R * W substitutions, giving about 52 * 8 + 8 + 52 * 8 ~ 840 candidate strings for English. Each must be looked up individually. This is feasible but less elegant.

**Space comparison for 50,000 words:**
- R-way trie (R = 128 ASCII): Up to 128 pointers * ~400,000 nodes = 51.2 million pointers. At 8 bytes each, this is ~400 MB. Excessive.
- TST: ~3 pointers * ~400,000 nodes = 1.2 million pointers. At 8 bytes each, ~10 MB. Very reasonable.
- Hash table: ~50,000 entries with overhead, perhaps 2-5 MB.

**Recommendation: TST.** It provides efficient performance for all three operations, uses reasonable space (much less than an R-way trie), supports the prefix-based operations that hash tables cannot, and can be adapted for fuzzy matching. The slight time overhead compared to hash tables for exact lookup (O(W + log N) vs. O(W)) is negligible for 50,000 words.

---

## Problem 7: DP Problem Identification

**Problem:** For each problem below, identify whether it has optimal substructure and overlapping subproblems (and thus is amenable to DP). If so, sketch the recurrence.

**(a) Minimum number of coins to make change for amount A, given coin denominations d_1, ..., d_k (unlimited supply of each).**

**Answer:** Yes, this is a classic DP problem (the "Coin Change" problem).
- **Optimal substructure:** If the optimal solution for amount A uses coin d_i as the last coin, then the remaining amount A - d_i must also be solved optimally. Any suboptimal solution for A - d_i could be improved, contradicting the optimality of the overall solution.
- **Overlapping subproblems:** Different sequences of coin choices lead to the same remaining amounts. For

example, with coins {1, 3, 4}, reaching amount 5 can happen via 4+1 or 3+1+1 or ..., and amount 5 must be solved regardless of how we arrived at it.
- **Recurrence:** dp[a] = min over all d_i <= a of (1 + dp[a - d_i]), with base case dp[0] = 0. dp[a] = infinity if no valid combination exists.
- **Time:** O(A * k). **Space:** O(A).

**(b) Finding the maximum element in an unsorted array.**

**Answer:** This problem does NOT benefit from DP. While it has optimal substructure in a trivial sense (the max of the full array is the max of the maxes of two halves), it has no overlapping subproblems. Each element and each subarray is examined exactly once. The problem is solved in O(N) time by a simple linear scan. No memoization or tabulation is needed or helpful.

**(c) Edit distance (Levenshtein distance) between two strings S1 and S2.**

**Answer:** Yes, this is a classic DP problem.
- **Subproblem:** dp[i][j] = minimum edit distance between S1[0..i-1] and S2[0..j-1].
- **Recurrence:**
- If S1[i-1] == S2[j-1]: dp[i][j] = dp[i-1][j-1] (characters match, no edit needed)
- Else: dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])
- dp[i-1][j] + 1: delete from S1
- dp[i][j-1] + 1: insert into S1
- dp[i-1][j-1] + 1: substitute
- **Base cases:** dp[i][0] = i (delete all i characters), dp[0][j] = j (insert all j characters).
- **Time:** O(N * M). **Space:** O(N * M), reducible to O(min(N, M)).

---

## Problem 8: Trie Delete Operation Trace

**Problem:** Given a trie containing the keys "the", "then", "there", and "them", trace the deletion of "then" and show the resulting trie.

**Solution:**

**Initial trie structure:**

```
root -> [t] -> [h] -> [e]* ("the") -> [n]* ("then")
                                   -> [r] -> [e]* ("there")
                                   -> [m]* ("them")
```

Node for 'e' (at depth 3) has value = "the" and three children: [n], [r], [m].

**Deleting "then" -- calling DeleteHelper(root, "then", 0):**

1. depth=0: key[0]='t', recurse on children['t'] with depth=1
2. depth=1: key[1]='h', recurse on children['h'] with depth=2
3. depth=2: key[2]='e', recurse on children['e'] with depth=3
4. depth=3: key[3]='n', recurse on children['n'] with depth=4
5. depth=4 == length("then"): Set node_n.value = NULL

Now check node_n: value is NULL. Check children: node_n has no children (it was a leaf).
Return NULL (delete this node).

1. Back at depth=3 (node_e): children['n'] = NULL (deleted).
   Check node_e: value is not NULL ("the"). KEEP this node. Return node_e.

2. The recursion unwinds without further deletions since node_e was kept.

**Resulting trie:**

```
root -> [t] -> [h] -> [e]* ("the") -> [r] -> [e]* ("there")
                                   -> [m]* ("them")
```

The node for 'n' was removed because after clearing its value, it had no value and no children. The node for 'e' at depth 3 was kept because it still has a value ("the") and two remaining children ([r] for "there" and [m] for "them"). The deletion was clean: only the leaf node was removed, and no other structure was affected.

# Summary

## Master Table: All Algorithms from the Complete Course

| Algorithm / Structure | Problem | Time (Average) | Time (Worst) | Space | Lecture |
|---|---|---|---|---|---|
| **Sorting** | | | | | |
| Selection Sort | Sort | O(N^2) | O(N^2) | O(1) | 3 |
| Insertion Sort | Sort | O(N^2) | O(N^2) | O(1) | 3 |
| Shellsort | Sort | O(N^{4/3}) est. | depends on gap sequence | O(1) | 3 |
| Mergesort | Sort | O(N log N) | O(N log N) | O(N) | 4 |
| Quicksort | Sort | O(N log N) | O(N^2) | O(log N) avg stack | 4 |
| Heapsort | Sort | O(N log N) | O(N log N) | O(1) | 7 |
| **Searching / Symbol Tables** | | | | | |
| Binary Search | Sorted array search | O(log N) | O(log N) | O(1) | 1 |
| BST (unbalanced) | Ordered symbol table | O(log N) | O(N) | O(N) | 5 |
| Red-Black BST | Ordered symbol table | O(log N) | O(log N) | O(N) | 5 |
| Hash Table (chaining) | Unordered symbol table | O(1) | O(N) | O(N) | 6 |
| Hash Table (probing) | Unordered symbol table | O(1) | O(N) | O(N) | 6 |
| R-Way Trie | String symbol table | O(W) | O(W) | O(R*N*W) | 12 |
| Ternary Search Trie | String symbol table | O(W + log N) | O(W + N) | O(N*W) | 12 |

| Algorithm / Structure | Problem | Time (Average) | Time (Worst) | Space | Lecture |
|---|---|---|---|---|---|
| **Priority Queues** | | | | | |
| Binary Heap | Insert / Delete-min | O(log N) | O(log N) | O(N) | 7 |
| **Graph Algorithms** | | | | | |
| Depth-First Search | Graph traversal | O(V + E) | O(V + E) | O(V) | 8 |
| Breadth-First Search | Shortest path (unwtd) | O(V + E) | O(V + E) | O(V) | 8 |
| Topological Sort | DAG ordering | O(V + E) | O(V + E) | O(V) | 8 |
| Dijkstra (binary heap) | SSSP (non-neg weights) | O(E log V) | O(E log V) | O(V) | 9 |
| Bellman-Ford | SSSP (any weights) | O(VE) | O(VE) | O(V) | 9 |
| Kruskal | Minimum spanning tree | O(E log E) | O(E log E) | O(E + V) | 10 |
| Prim (lazy) | Minimum spanning tree | O(E log E) | O(E log E) | O(E) | 10 |
| Prim (eager / indexed PQ) | Minimum spanning tree | O(E log V) | O(E log V) | O(V) | 10 |
| **Union-Find** | | | | | |
| Quick-Find | Dynamic connectivity | O(1) find | O(N) union | O(N) | 11 |
| Quick-Union | Dynamic connectivity | O(N) | O(N) | O(N) | 11 |
| Weighted QU + Path Comp. | Dynamic connectivity | O(alpha(N)) amort. | O(alpha(N)) amort. | O(N) | 11 |

| Algorithm / Structure | Problem | Time (Average) | Time (Worst) | Space | Lecture |
|---|---|---|---|---|---|
| **String Algorithms** | | | | | |
| Brute Force Search | Substring search | O(N) typical | O(NM) | O(1) | 12 |
| KMP (LPS array) | Substring search | O(N + M) | O(N + M) | O(M) | 12 |
| KMP (DFA) | Substring search | O(N + RM) build | O(N + RM) | O(RM) | 12 |
| **Dynamic Programming** | | | | | |
| Fibonacci (DP) | Nth Fibonacci number | O(N) | O(N) | O(1) optimized | 12 |
| Longest Common Subseq. | Sequence alignment | O(NM) | O(NM) | O(NM) or O(min(N,M)) | 12 |
| 0/1 Knapsack | Combinatorial optimization | O(NW) | O(NW) | O(NW) or O(W) | 12 |

## Key Takeaways from the Entire Course

1. **Algorithm analysis is the foundation.** Without Big-O notation and careful operation counting, we cannot meaningfully compare solutions or predict performance. Every design decision should be backed by complexity analysis.

2. **Sorting is the most well-studied problem in computer science.** The O(N log N) lower bound for comparison-based sorting is a fundamental limit. Among comparison sorts, quicksort reigns in practice due to its cache-friendliness and small constant factors, despite its O(N^2) worst case.

3. **Trees are the most versatile data structure family.** BSTs support ordered operations in O(log N) time. Heaps support priority operations. Tries support string-specific operations. Balanced variants (red-black trees) provide worst-case guarantees. Trees appear everywhere.

4. **Hashing provides O(1) expected time for point queries but sacrifices ordering.** The choice between hash tables and balanced BSTs is one of the most common design decisions. If you need only insert/

search/delete, use a hash table. If you need min/max, rank, range queries, or ordered iteration, use a balanced BST.

5. **Graphs model an enormous range of real-world problems.** BFS, DFS, Dijkstra, Bellman-Ford, Kruskal, and Prim are indispensable tools. Learning to recognize when a problem is a graph problem in disguise is a critical algorithmic skill.

6. **Dynamic programming transforms exponential problems into polynomial ones.** The systematic DP methodology -- define subproblems, write the recurrence, identify base cases, determine evaluation order -- provides a structured approach to problems that initially seem intractable.

7. **There is no single best algorithm or data structure.** The right choice depends on the problem constraints, input characteristics, operation mix, and practical requirements. Understanding tradeoffs is the hallmark of algorithmic expertise, and the true goal of a course like this one.

---

## References

1. Sedgewick, R. & Wayne, K. (2011). *Algorithms*, 4th Edition. Addison-Wesley. Chapters 5.1 (String Sorts), 5.2 (Tries), 5.3 (Substring Search). The primary text for this course.

2. Knuth, D.E., Morris, J.H., & Pratt, V.R. (1977). "Fast Pattern Matching in Strings." *SIAM Journal on Computing*, 6(2), 323-350. The original KMP paper.

3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., & Stein, C. (2009). *Introduction to Algorithms*, 3rd Edition. MIT Press. Chapters 15 (Dynamic Programming), 32 (String Matching). The standard algorithms reference.

4. Bellman, R.E. (1957). *Dynamic Programming*. Princeton University Press. The foundational monograph on DP by its inventor.

5. Boyer, R.S. & Moore, J.S. (1977). "A Fast String Searching Algorithm." *Communications of the ACM*, 20(10), 762-772. The Boyer-Moore algorithm for practical substring search.

6. Karp, R.M. & Rabin, M.O. (1987). "Efficient Randomized Pattern-Matching Algorithms." *IBM Journal of Research and Development*, 31(2), 249-260. The Rabin-Karp rolling hash approach.

7. Bentley, J.L. & Sedgewick, R. (1997). "Fast Algorithms for Sorting and Searching Strings." *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 360-369. Introduces ternary search tries.

8. Fredkin, E. (1960). "Trie Memory." *Communications of the ACM*, 3(9), 490-499. The original paper introducing the trie data structure.

9. Skiena, S.S. (2008). *The Algorithm Design Manual*, 2nd Edition. Springer. An excellent practical companion with war stories and problem-solving heuristics.

10. Kleinberg, J. & Tardos, E. (2005). *Algorithm Design*. Pearson. Outstanding coverage of dynamic programming, network flow, and NP-completeness. Excellent complement to Sedgewick.

---

*Lecture 12 of 12* -- *Data Structures & Algorithms* -- *Based on Sedgewick's Algorithms, 4th Edition*