CS 589 Lecture 4 Study Guide: Information Retrieval Infrastructure

📚 Table of Contents

Overview & Learning Objectives {#overview}

What You Should Know After This Lecture:

- ✅ Why inverted indexes are necessary for efficient IR
- ✅ How to build an inverted index from documents
- ✅ How to process Boolean queries (AND, OR, NOT)
- ✅ Linear time merge algorithm and its complexity
- ✅ Compression techniques (28N → 11N → 9.75N)
- ✅ Prefix encoding (gamma codes)
- ✅ Trie data structures and time complexity
- ✅ Vector databases and HNSW

---

1. Inverted Index Fundamentals {#fundamentals}

1.1 Motivation: The Scalability Challenge

Problem:

- Consider N = 1 million documents, each with ~1000 words
- Average 6 bytes/word = 6GB of data
- M = 500K distinct terms
- A term-document matrix would have 500K × 1M = 500 billion entries!

- But only ~1 billion 1's (most entries are 0)

Solution: Only store the 1's using an **inverted index**

1.2 What is an Inverted Index?

An inverted index is a data structure that maps each term to a list of documents containing that term.

Structure:

```
Dictionary (Terms) → Postings Lists (Document IDs)

Brutus  → [1, 2, 4, 11, 31, 45, 173, 174]
Caesar  → [1, 2, 4, 5, 6, 16, 57, 132]
Calpurnia → [2, 31, 54, 101]
```

Key Properties:

- Posting lists are **sorted by document ID** (critical for efficient merging!)
- Each term stores its **document frequency** (DF)
- Much more space-efficient than term-document matrix

1.3 Example: Shakespeare Query

Query: "Which plays contain Brutus AND Caesar but NOT Calpurnia?"

Using Boolean operations:

```
Brutus:    110100... (binary representation)
Caesar:    110111...
~Calpurnia: 101111... (NOT operation)
Result:    100100... (Antony and Cleopatra, Hamlet)
```

---

2. Building an Inverted Index {#building}

2.1 The Indexing Pipeline

```
Documents → Tokenizer → Token Stream → Linguistic Modules →
Modified Tokens → Indexer → Inverted Index
```

Example:

```
Input: "Friends, Romans, countrymen."

↓ Tokenizer
Tokens: [Friends, Romans, Countrymen]

↓ Linguistic Modules (lowercase, stemming)
Modified: [friend, roman, countryman]

↓ Indexer
Index:
  friend → [2, 4]
  roman → [1, 2]
  countryman → [13, 16]
```

2.2 Text Preprocessing Steps

1. Tokenization

- Cut character sequences into word tokens
- Handle cases like "John's", "state-of-the-art"

2. Normalization

- Map variants to same form
- Example: "U.S.A." → "USA"

3. Stemming

- Reduce words to root form
- Example: "authorize" → "author", "authorization" → "author"

4. Stop words (optional)

- May omit very common words: the, a, to, of

2.3 Index Construction Algorithm

Step 1: Create (term, docID) pairs

```
Doc 1: "I did enact Julius Caesar..."
Doc 2: "So let it be with Caesar..."

Pairs:
(I, 1), (did, 1), (enact, 1), (julius, 1), (caesar, 1)...
(so, 2), (let, 2), (it, 2), (be, 2), (caesar, 2)...
```

Step 2: Sort by term, then by docID

```python
# Python sorting with lambda
sorted(pairs, key=lambda x: (x[0], x[1]))
```

Result:

```
(ambitious, 2), (be, 2), (brutus, 1), (brutus, 2),
(capitol, 1), (caesar, 1), (caesar, 2)...
```

Step 3: Merge and create postings

```
Term      DF  → Postings
brutus    2   → [1, 2]
caesar    2   → [1, 2]
capitol   1   → [1]
```

---

3. Query Processing {#query-processing}

3.1 Boolean Query Processing

Query: Brutus AND Caesar

Algorithm:

1. Locate "Brutus" in dictionary → retrieve postings
2. Locate "Caesar" in dictionary → retrieve postings
3. Merge (intersect) the two posting lists

3.2 Linear Time Merge Algorithm

Time Complexity: O(x + y) where x, y are posting list lengths

```python
```

```
def intersect(p1, p2):
    """
    Merge two sorted posting lists
    """
    answer = []
    i, j = 0, 0

    while i < len(p1) and j < len(p2):
        if p1[i] == p2[j]:
            answer.append(p1[i])
            i += 1
            j += 1
        elif p1[i] < p2[j]:
            i += 1
        else:
            j += 1

    return answer
```

Example:

```
Brutus:  [2, 4, 8, 16, 32, 64, 128]
Caesar:  [1, 2, 3, 5, 8, 13, 21, 34]

Step by step:
i→2, j→1: 2>1, j++
i→2, j→2: 2==2, add 2, i++, j++
i→4, j→3: 4>3, j++
i→4, j→5: 4<5, i++
i→8, j→5: 8>5, j++
i→8, j→8: 8==8, add 8, i++, j++

Result: [2, 8]
```

3.3 Handling OR and NOT

OR Operation: Similar to AND, but include all elements from both lists.

NOT Operation:

```
Brutus AND NOT Caesar

1. Get Brutus postings: [2, 4, 8, 16, 32, 64, 128]
2. Get Caesar postings: [1, 2, 3, 5, 8, 13, 21, 34]
3. Remove Caesar docs from Brutus list
Result: [4, 16, 32, 64, 128]
```

3.4 Query Optimization

Problem: For multi-term AND queries, which order should we process terms?

Answer: Process in order of **increasing frequency** (smallest posting list first)

Example:

```
Query: Brutus AND Calpurnia AND Caesar

Document Frequencies:
- Calpurnia: 4 docs
- Brutus: 8 docs
- Caesar: 8 docs

Optimal order: (Calpurnia AND Brutus) AND Caesar
```

Why? Start with smallest set, then keep cutting it down!

---

4. Query Time Optimization {#query-optimization}

4.1 Skipping Lists

Problem: Can we skip comparing some elements during merge?

Solution: Add skip pointers every k elements

```
Brutus: 2→4→8→16→19→23→28→43
        ↓  ↓     ↓
       16  28    72
```

How it works: When comparing, if skip pointer target < current element in other list, jump ahead!

Example:

```
Brutus: [2, 4, 8, 16, 19, 23, 28, 43]
        skip to 16, skip to 28
Caesar: [1, 2, 3, 5, 8, 41, 51, 60, 71]
        skip to 5, skip to 51

When at Brutus[2] and Caesar[1]:
- 2 > 1, check Caesar's skip to 5
- 5 > 2, so jump Caesar to position 5
- Continue merging...
```

4.2 Proximity Search

Goal: Find documents where terms appear near each other

Query: "to be or not to be" within 5 words

Solution: Store **positions** in postings

```
Positional Index:
to:  <2: [1,17,74,222,551], 4: [8,16,190,429,433], ...>
be:  <1: [17,19], 4: [17,191,291,430,434], ...>
or:  <1: [2], 2: [8], ...>
not: <1: [3], 2: [1], ...>
```

Example Document:

```
d1: "To be or not to be, that is the question."

Positional postings:
to:  [0, 4]
be:  [1, 5]
or:  [2]
not: [3]
that: [6]
```

Proximity Algorithm:

1. Get all position lists
2. Check if positions satisfy proximity constraint
3. Use sliding window approach

---

5. Index Compression {#compression}

5.1 Compressing the Dictionary (Posting Pointer Table)

Initial storage: 28N bytes

- 20 bytes for term string
- 4 bytes for document frequency
- 4 bytes for posting pointer

Optimization 1: Concatenate dictionary as one string

- Storage: 28N → 11N bytes
- Store: frequency (4 bytes) + term pointer (4 bytes) + posting pointer (4 bytes) = 12 bytes + term length

Optimization 2: Store gaps instead of absolute pointers

- Skip k-1 pointers for every k pointers
- Recover skipped pointers by adding word lengths
- Storage: 11N → 9.75N (when k=4)

Formula: $8N + 3N \times ((3+k)/(3\times k))$

- When k=4: $8N + 3N \times (7/12) = 9.75N$

5.2 Compressing Posting Lists

Key Observation: Store gaps instead of absolute docIDs

```
Original: [2, 4, 8, 16, 19, 23, 28, 43]
Gaps:     [2, 2, 4, 8, 3, 4, 5, 15]
```

Binary representation of gaps:

```
Gaps:   [2,  4,  8,  3,  4,  5,  15]
Binary: [10, 100, 1000, 11, 100, 101, 1111]
```

5.3 Prefix Encoding (Gamma Codes)

Problem: Need uniquely decodable encoding

Gamma Code Structure:

- Length component: unary code for (length - 1)
- Offset component: last (length - 1) bits of binary

Example: Encode 13

13 in binary: 1101 (4 bits)
Length - 1 = 3

Gamma code:
- Length: 1110 (three 1's + 0)
- Offset: 101 (last 3 bits of 1101)
- Result: 1110,101

Practice: Encode 5

5 in binary: 101 (3 bits)
Length - 1 = 2

Gamma code:
- Length: 110 (two 1's + 0)
- Offset: 01 (last 2 bits of 101)
- Result: 110,01

Decoding:

Sequence: 1110101110001...

1. Count 1's until you hit 0: 1110 → 3 ones → length = 4
2. Read next 3 bits: 101
3. Reconstruct: 1 + 101 = 1101 = 13
4. Continue: 110 → 2 ones → length = 3...

Quiz Question from Lecture 5: Q: Which are prefix encodings?

- b, c are prefix encodings ✓
- Gamma code is a prefix encoding

Q: What are the gamma codes for [77, 87, 49, 50]?

```
77 = 1001101 (7 bits) → length-1 = 6
  → 1111110,001101

87 = 1010111 (7 bits) → length-1 = 6
  → 1111110,010111

49 = 110001 (6 bits) → length-1 = 5
  → 111110,10001

50 = 110010 (6 bits) → length-1 = 5
  → 111110,10010
```

5.4 Compression Results

Compression rate: 11.7% (gamma codes)

---

6. Dictionary Search Optimization {#dictionary-optimization}

6.1 Trie (Prefix Tree)

Time Complexity Comparison:

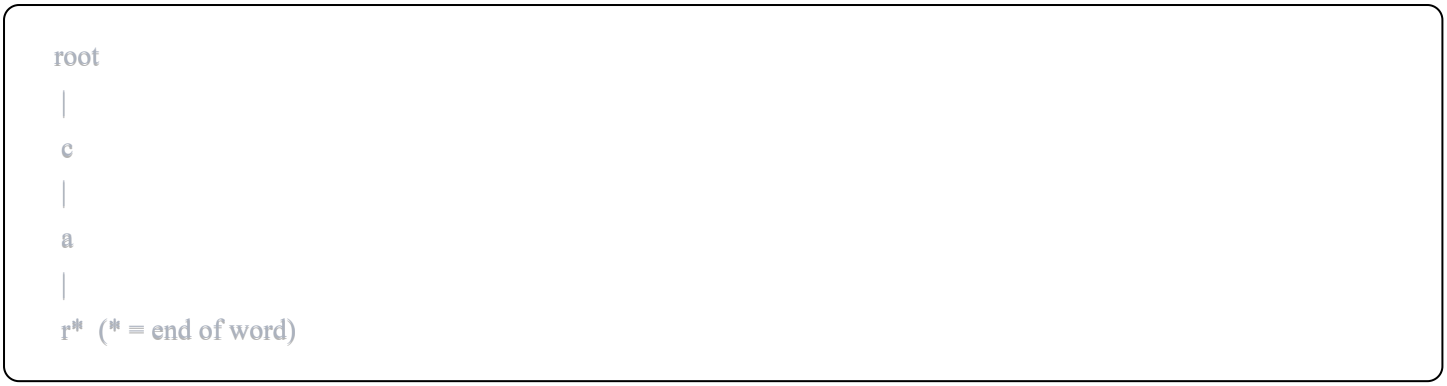
- Binary Search Tree: O(m × log n)
  - m = maximum word length
  - n = vocabulary size
- Trie: **O(m)** - only depends on word length!

Trie Structure:

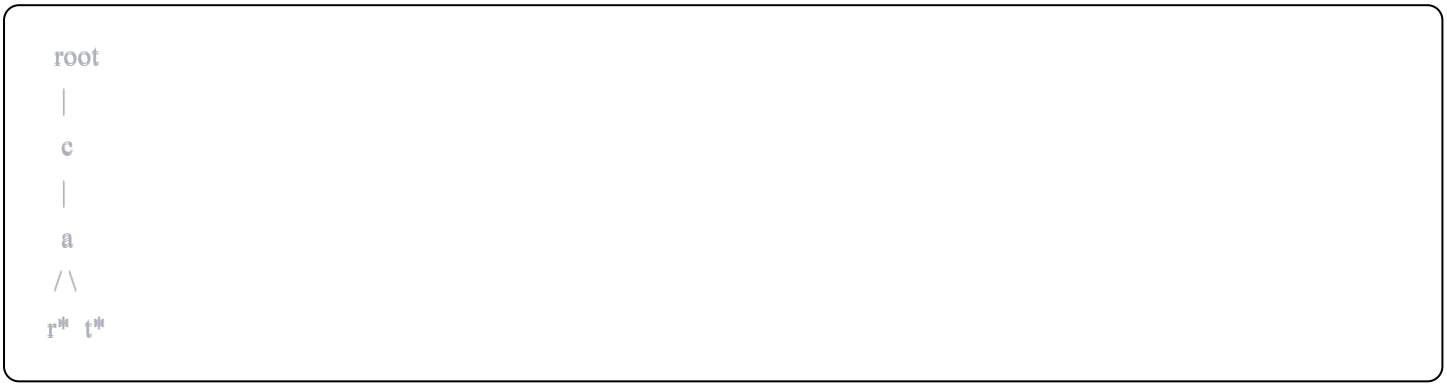
Words: car, cat, do, don

Building a Trie:

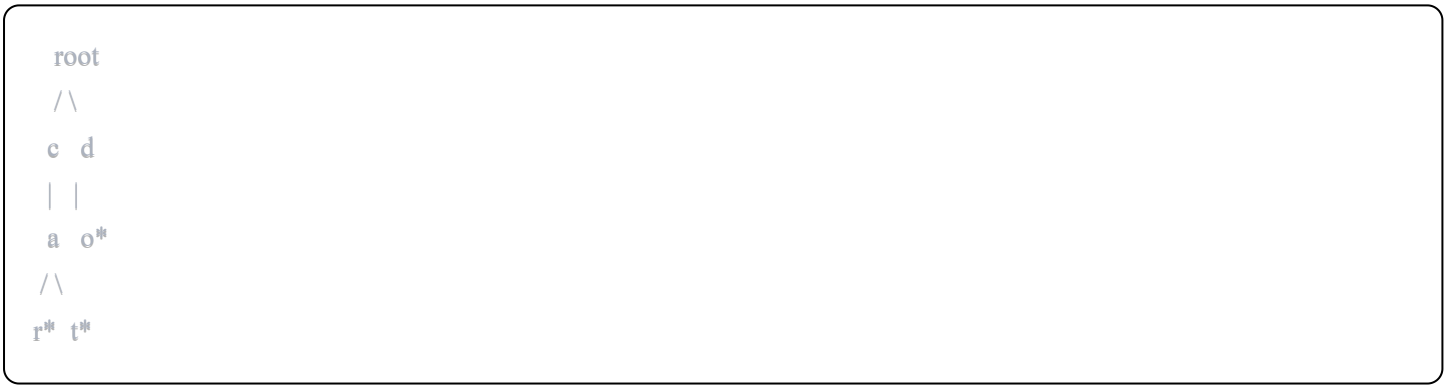
Step 1: Insert "car"



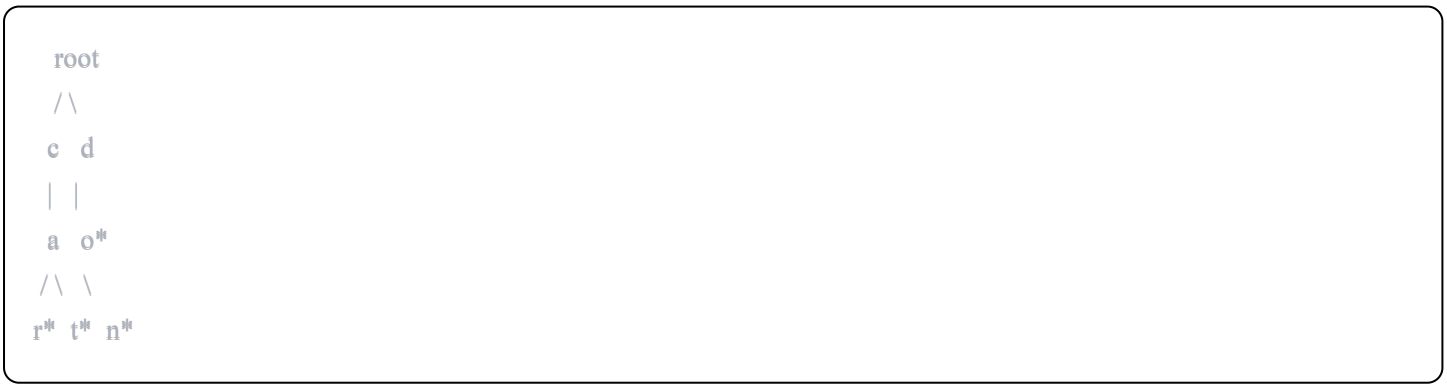
Step 2: Insert "cat"



Step 3: Insert "do"



Step 4: Insert "don"



Search Example: Find "cat"

1. Start at root
2. Follow 'c' → found
3. Follow 'a' → found

4. Follow 't' → found, marked as end
5. Time: O(3) = O(m)

---

7. Advanced Topics {#advanced}

7.1 Vector Databases & HNSW

Hierarchical Navigable Small World (HNSW)

Problem: Nearest neighbor search in high-dimensional spaces

Key Ideas:

1. **Small World Networks:** Most nodes are few hops away
2. **Hierarchical Structure:** Multiple layers for coarse-to-fine search
3. **Navigable:** Use greedy search to find closest points

HNSW Structure:

```
Layer 3 (top): Few entry points, long-range connections
  ↓
Layer 2: More nodes, medium-range connections
  ↓
Layer 1: Even more nodes, short-range connections
  ↓
Layer 0 (bottom): All nodes, nearest neighbors
```

Layer Assignment:

- Use exponential decay probability
- P[level] = func(level, mL)
- Higher layers: fewer nodes, longer jumps
- Lower layers: more nodes, finer search

Search Algorithm:

1. Enter at top layer
2. Greedily move to closer neighbors
3. When can't get closer, drop to next layer
4. Repeat until bottom layer
5. Return k nearest neighbors

Parameters:

- M: number of connections per node
- M_max: max connections for layers > 0
- M_max0: max connections for layer 0
- Typically M_max0 > M_max for better connectivity at base

Optimal Values:

- k = 3 × d (where d is dimensionality)
- w changes dynamically: A × log(n_current)

7.2 MapReduce for Large-Scale Indexing

Problem: Web-scale indexing requires distributed processing

Google's Solution: MapReduce

Architecture:

```
Documents → Master → Parser (Map) → Segment Files →
Inverter (Reduce) → Postings
```

Map Phase:

- Input: Document splits
- Output: (term, docID) pairs
- Example: "Caesar died" → [(Caesar, d1), (died, d1)]

Reduce Phase:

- Input: (term, [docID list])
- Output: Inverted index entries
- Groups: a-f, g-p, q-z (term partitions)

Example:

```
Map: d2 contains "C died", d1 contains "C came, C c'ed"

Pairs:
(C, d2), (died, d2), (C, d1), (came, d1),
(C, d1), (c'ed, d1)

After sorting:
(C, [d1, d1, d2])
(came, [d1])
(c'ed, [d1])
(died, [d2])

Reduce:
C     → [d1:2, d2:1]
came  → [d1:1]
c'ed  → [d1:1]
died  → [d2:1]
```

Partitioning Strategies:

1. **Term-partitioned:** Each machine handles term range
2. **Document-partitioned:** Each machine handles document range

Industry Practice: Document-partitioned (better load balancing!)

---

8. Practice Problems {#practice}

Problem 1: Build an Inverted Index

Given documents:

```
d1: "cat dog cat"
d2: "dog bird"
d3: "cat bird dog bird"
```

Solution:

Step 1: Create (term, docID) pairs

```
(cat,d1), (dog,d1), (cat,d1),
(dog,d2), (bird,d2),
(cat,d3), (bird,d3), (dog,d3), (bird,d3)
```

Step 2: Sort

```
(bird,d2), (bird,d3), (bird,d3),
(cat,d1), (cat,d1), (cat,d3),
(dog,d1), (dog,d2), (dog,d3)
```

Step 3: Create index

```
Term   DF → Postings
bird   2  → [d2, d3]
cat    2  → [d1, d3]
dog    3  → [d1, d2, d3]
```

Problem 2: Query Processing

Query: cat AND dog

Solution:

```
cat: [d1, d3]
dog: [d1, d2, d3]

Merge:
i→d1, j→d1: match! Add d1
i→d3, j→d2: d3>d2, j++
i→d3, j→d3: match! Add d3

Result: [d1, d3]
```

Problem 3: Query Optimization

Query: (tangerine OR trees) AND (marmalade OR skies) AND (kaleidoscope OR eyes)

Term frequencies:

```
eyes:         213,312
kaleidoscope:  87,009
marmalade:    107,913
skies:        271,658
tangerine:     46,653
trees:        316,812
```

Solution:

Process OR clauses in order of their **combined** frequency:

1. (kaleidoscope OR eyes) = 87,009 + 213,312 = 300,321
2. (marmalade OR skies) = 107,913 + 271,658 = 379,571

3. (tangerine OR trees) = 46,653 + 316,812 = 363,465

Optimal order:

```
(kaleidoscope OR eyes) AND
(tangerine OR trees) AND
(marmalade OR skies)
```

Problem 4: Gamma Code Encoding

Encode the number 24:

```
24 in binary: 11000 (5 bits)
Length - 1 = 4

Gamma code:
- Length: 11110 (four 1's + 0)
- Offset: 1000 (last 4 bits of 11000)
- Result: 11110,1000
```

Problem 5: Trie Construction

Build a trie for: ["cat", "car", "card", "care", "can", "dog", "door"]

Solution:

Time to search "card": O(4) = O(length)

📝 Quiz Solutions from Lecture 4

Quiz Question 1: MAP Calculation

Given:

- System A: [+,+,−,−,...]
- System B: [+,−,+,−,...]
- 100 documents total, 4 relevant

Answer: 1/2, 5/12

Calculation:

```
System A:
P@1 = 1/1 = 1
P@2 = 2/2 = 1
No more relevant docs
MAP_A = (1 + 1)/2 = 1/2

System B:
P@1 = 1/1 = 1
P@3 = 2/3
No more relevant docs
MAP_B = (1 + 2/3)/2 = 5/12
```

Quiz Question 2: NDCG Calculation

Answer: 0.5855

Formula:

```python
import math

# DCG = (1 + 1/2) / (1 + 1/log₂(3) + 1/2 + 1/log₂(5))
numerator = 1 + 0.5
denominator = 1 + 1/math.log2(3) + 0.5 + 1/math.log2(5)
result = numerator / denominator
# Result: 0.5855
```

🎯 Key Takeaways

1. **Inverted Index = Dictionary + Postings Lists**
   - Sorted by docID for efficient merging
   - Stores document frequency for query optimization
2. **Query Processing Time: O(x + y)**
   - Linear merge for AND queries

- Process terms in increasing frequency order

3. **Compression Techniques:**
   - Dictionary: 28N → 11N → 9.75N
   - Postings: Store gaps + gamma encoding → 11.7% compression
4. **Trie Search: O(m)** where m = word length
   - Independent of vocabulary size!
5. **HNSW for Vector Search:**
   - Hierarchical layers for coarse-to-fine search
   - Logarithmic complexity with right parameters
6. **MapReduce for Scale:**
   - Document-partitioned indexing
   - Better load balancing than term-partitioned

📺 Recommended Videos

1. **Inverted Index Basics:**
   - Stanford CS276: Inverted Index
2. **HNSW Explained:**
   - James Briggs: HNSW for Vector Search
   - Pinecone: Understanding HNSW
3. **Gamma Codes:**
   - Variable Length Encoding
4. **MapReduce:**
   - Google MapReduce Paper Explained

🔗 Additional Resources

- **Stanford IR Book:** https://nlp.stanford.edu/IR-book/
  - Chapter 5: Index compression
  - Chapter 4: Index construction
- **ElasticSearch Tutorial:**
  - Official docs: https://www.elastic.co/guide/
  - Hands-on: Build your own search engine

- **FAISS (Facebook AI Similarity Search):**
  - GitHub: https://github.com/facebookresearch/faiss
  - Tutorial: https://www.pinecone.io/learn/series/faiss/

Good luck on your midterm! 🍀

Remember: Practice building indices by hand, work through the merge algorithm examples, and understand WHY each optimization works, not just HOW!