Q1) What is "parsing" in NLP?

Definition.
Parsing is the process of assigning a **syntactic structure** to a sentence according to a grammar. The output is typically a **parse tree** (constituency parsing) or a **dependency graph** (dependency parsing). The goal is to determine how words group into phrases and how those phrases relate, enabling downstream understanding.

Why it matters.

- **Disambiguation:** resolves multiple possible structures ("I saw the man with a telescope").
- **Interfaces to meaning:** syntactic structure is a scaffold for **semantic role labeling**, **information extraction**, and **question answering**.
- **Downstream utility:** improves **machine translation**, **ASR post-processing**, **IE**, **coreference**, and **code generation**.

Outputs.

- **Constituency parse:** hierarchical phrases (e.g., `S → NP VP`).
- **Dependency parse:** head–dependent relations (e.g., `nsubj(sleeps, cat)`).

Q2) Key steps of the CKY (Cocke–Kasami–Younger) algorithm

> Dynamic-programming parser for **context-free grammars in Chomsky Normal Form (CNF)**.

1. **CNF conversion**
   Convert grammar so every rule is either `A → BC` or `A → a`. (Add ε/UNIT-rule handling before conversion.)
2. **Initialization (length-1 spans)**
   For each token `w_i`, fill cell `(i,i+1)` with all nonterminals `A` such that `A → w_i`.
3. **Bottom-up chart filling (longer spans)**
   For span length `ℓ = 2..n`, for start `i`, let `j = i+ℓ`. For every split `k` with `i < k < j`, and rules `A → B C`, if `B ∈ chart[i,k]` and `C ∈ chart[k,j]`, then **add** `A` **to** `chart[i,j]` (store backpointers for reconstruction).
4. **Backpointer reconstruction**
   Recover the highest-scoring or all valid trees from backpointers in `chart[0,n]` that produce the start symbol (usually `S`).
5. **(Optional) Probabilistic scoring**
   With **PCFG** rule probabilities, compute **inside scores** while filling; select the **Viterbi** best tree.

Complexity: `O(n³ · |G|)` time, `O(n² · |N|)` space (n = sentence length).

Q3) How PCFGs differ from traditional CFGs

Aspect	CFG	PCFG
Rules	Production rules only	Production rules **with probabilities** `P(A→α)`

Aspect	CFG	PCFG
Ambiguity	Returns all parses	Returns **ranked parses**; enables **most-likely** parse (Viterbi)
Learning	Manually designed	**Estimated** from treebanks via relative frequency or **Inside–Outside (EM)**
Inference	Boolean membership	**Weighted** parsing (inside/outside probabilities)

Intuition: PCFGs capture **preference patterns** (e.g., `VP → V NP` more frequent than `VP → V PP`), improving robustness on ambiguous inputs.

Q4) Purpose of the Inside–Outside algorithm in PCFG training

Goal.
Estimate **maximum-likelihood rule probabilities** `P(A→α)` from **unlabeled or partially labeled** data when gold parse trees aren't available.

How it works (EM over trees):

- **E-step (Expectation):**
  For each sentence, run dynamic-programming to compute:
  - **Inside** probabilities `β(i,j,A)` = prob that `A` generates span `(i,j)`.
  - **Outside** probabilities `α(i,j,A)` = prob of the context around that span.
    Combine them to get **expected counts** of rule uses over all latent parses.
- **M-step (Maximization):**
  Update rule probabilities by normalizing expected counts:

$$\hat{P}(A \to \alpha) = \frac{\mathrm{E}[\#(A \to \alpha)]}{\sum_{\alpha'} \mathrm{E}[\#(A \to \alpha')]}.$$

Outcome: Iteratively improves PCFG parameters to better explain the training sentences (monotonic likelihood increase under EM).

Q5) Apply CKY to "the cat sleeps" with a simple CFG

Sentence: `the / cat / sleeps` (n = 3)

CNF Grammar (toy):

```
S   → NP VP
NP  → Det N
VP  → V
Det → the
```

```
N   → cat
V   → sleeps
```

Chart indices: tokens at positions 0..3
Cells are spans `(i,j)` over half-open intervals.

Initialization (length 1):

- `(0,1)` : `Det` (because `Det→the`)
- `(1,2)` : `N` (because `N→cat`)
- `(2,3)` : `V` (because `V→sleeps`)

Length 2 spans:

- `(0,2)` with split `k=1` : we have `Det` in `(0,1)` and `N` in `(1,2)` ; rule `NP → Det N` fires → add `NP` .
- `(1,3)` with split `k=2` : we have `N` and `V` , but no rule `X → N V` → **no addition**.

Length 3 span (0,3):

- Try splits:
  - `k=1` : `(0,1)` = `Det` , `(1,3)` =∅ → no rule.
  - `k=2` : `(0,2)` = `NP` , `(2,3)` = `V` ; rule `VP → V` is unary, so first ensure `(2,3)` includes `VP` (via `VP→V`). Then `S → NP VP` with `(0,2)=NP` and `(2,3)=VP` → add `S` .

Result: `chart[0,3]` contains `S` .
Parse tree (constituency):

```
        S
      /   \
    NP     VP
   / \      |
Det   N     V
 |    |     |
the cat   sleeps
```

Exam tip: If your version of CKY strictly disallows unary rules inside the DP step, pre-convert unary `VP→V` by binarization or handle unary-closure per cell after fills.

Q6) Demonstrate transition-based parsing with SHIFT/REDUCE

Below are two quick demonstrations—one **constituency** (classic shift-reduce) and one **dependency** (arc-standard). In exams, state which paradigm you're using.

A) Constituency shift-reduce (for the same sentence)

Sentence: "the cat sleeps"
Goal tree: `S → NP VP` , `NP→ Det N` , `VP → V`

Actions and configurations (Stack | Buffer):

1. `[] | the cat sleeps` — **SHIFT** → `[the] | cat sleeps`
2. `[the] | cat sleeps` — **SHIFT** → `[the, cat] | sleeps`
3. `[the, cat] | sleeps` — **REDUCE→N** (using `N→cat`) → `[the, N] | sleeps`
4. `[the, N] | sleeps` — **REDUCE→Det** (using `Det→the`) → `[Det, N] | sleeps`
5. `[Det, N] | sleeps` — **REDUCE→NP** (`NP→Det N`) → `[NP] | sleeps`
6. `[NP] | sleeps` — **SHIFT** → `[NP, sleeps] |`
7. `[NP, sleeps] |` — **REDUCE→V** (`V→sleeps`) → `[NP, V] |`
8. `[NP, V] |` — **REDUCE→VP** (`VP→V`) → `[NP, VP] |`
9. `[NP, VP] |` — **REDUCE→S** (`S→NP VP`) → `[S] | ∅` (accepted)

Notes.

- A practical system uses a learned classifier to decide whether to SHIFT or which REDUCE to apply.
- Unary reductions can be handled via separate **UNARY-REDUCE** or by applying closures.

B) Dependency parsing (arc-standard variant)

Target dependencies: `nsubj(sleeps, cat)`, `det(cat, the)`

Actions (Stack | Buffer | Arcs):

1. `[] | the cat sleeps | ∅` — **SHIFT** → `[the] | cat sleeps | ∅`
2. `[the] | cat sleeps` — **SHIFT** → `[the, cat] | sleeps | ∅`
3. `[the, cat] | sleeps` — **LEFT-ARC(det)** makes `det(cat, the)` and pops `the` → `[cat] | sleeps | {det(cat,the)}`
4. `[cat] | sleeps` — **SHIFT** → `[cat, sleeps] | ∅ | {…}`
5. `[cat, sleeps] | ∅` — **RIGHT-ARC(nsubj)** makes `nsubj(sleeps, cat)` and pops `cat` → `[sleeps] | ∅ | {det(cat,the), nsubj(sleeps,cat)}`
6. `[sleeps] | ∅` — **REDUCE** (or **ROOT-attach** depending on formalism) → done.

Takeaway.
Transition-based parsers use small, local actions to build structure incrementally; learned policies (e.g., perceptron, MLP, BiLSTM encoders, transformers) choose actions based on stack/buffer features.

Practical applications of parsing (for exam answers)

- **Information extraction:** Identify subject–verb–object triples from parsed structure for knowledge graph population.
- **Question answering:** Map wh-phrases to their governing predicates to extract precise answers.
- **Machine translation:** Syntactic constraints reduce reordering errors in phrase-based/Neural MT (syntax-aware models).
- **Text simplification & grammar checking:** Detect malformed structures and propose repairs.
- **Voice assistants:** Parse user commands to semantic frames (e.g., `Book → [Agent, Theme, Time]`).