Q1

(a) I have chosen Berkeley Neural Parser (BNP) to have some experiments. BNP is the most widely-used English parser for short sentence, and it is based on the neural network, achieved SOTA on corpus like Penn Treebank.

The parse trees generated are not always binary trees as we using CNF in the class. For example, when parsing "the only one", BNP directly parses it into "DET JJ(adj) NN" instead of "NP -> Det NP" and "NP -> JJ NN". To be more specific, the parse tree generted by BNP are more flatter and less hierarchical, and it normally has more direct constituents, where each node might have more children.

More surprisingly, for those sentences that are ambiguous, (e.g., prepositional attachment), BNP directly merge them together as in one rule, which enforce people to distinguish what exactly is the true structure by themselves.

(b)
(1) A special case is that if I ask it to parse a gramaticaly wrong sentence such as "I play the game eat some food", where "and" is ignored, and it can still parse this sentence and assign "VBP" (verb present) to "play" and "VB" to "eat", which means that BNP cannot detect that there is mistake within the sentence.
(2) BNP is not good at processing prepositional phrase attachment. For example, in the sentence "I saw the man in the park with a telescope", BNP choose to merge "saw" (VBD), "the man in the park" (NP), and "with a telescope" (PP) together, instead of assigning PP to VP (VBD+NP) or to NP first.
(3) BNP also deals bad with coordination ambiguity such as "I saw the old man and woman". It cannot decide to assign "old" to "man and woman" or only "man".
(4) BNP is not dealing well with polysemy. For example, "The old man the boat", where "man" is the verb, can be parsed into 2 NPs.

However, BNP deals very well with complex structures such as nested clauses and long dependency. For example, in the sentence "The cat that the dog, which was barking loudly, chased, was fast", BNP successfully assigned "fast" with "the cat".

(c)

1. "The more you push, the less you take."
   BNP parsed "more" as RBR (adverb comparative) and "less" as JJR (adjective comparative). But "more" should also be a JJR. This is a parsing mistake that caused by comparative correlative, a normal structure in English.
2. "I saw the man in the park with the telescope."
   BNP attached "the telescope" with "the man in the park" as a NP. But smantically, this sentence is more likely to be intepreted as "the man in the park" is a NP, and "with the telescope" is a PP, which applys to subject 'I'. This is an attachment ambiguity that caused by additional adjunct.
3. "Old men and teenagers are waiting outside."
   In this sentence, BNP couldn't decide to assign "old" to "the man" or "the man and teenagers". This is a NP bracketing ambiguity that the initial adjective could be applied to the first NP or all the NPs if the following ones does not have adjectives attached ahead.

Q2

(a)
[1] Dependency grammar does not generate a syntax tree, instead, it describes the structure of sentence by using the dependant relationship between each words. Every token is directly connected with its corresponding head token. There is no context-free nonterminals such as NP or VP. The syntax information is only represented by the dependency arcs. For example, nsubj represents the noun subject, and dobj represents direct object.
[2] Link grammar parser constructs a syntactic structure represented as links between pairs of words. Those links are named with the relationship btween words. For example, MV connects verbs (and adjectives) to modifying phrases like adverbs, prepositianl pjrases, time expressions, etc. By checking 'Show all linkages', all the possible linkages (syntactic structures) are shown, By checking 'Show constituent tree', it can generate trees (represented by parenthesis) as we saw in the HW1. However, this parser is not good at propernouns. If I input 'Papa ate the caviar with a spoon', it falls in 'No complete linkagesn found' because Papa is not in its grammar vocabulary.
[3] The link provided is not accessible, and there is no relative online demo. However, I have read many relative documents to have a thorough view of this parser. As opposed to dependency grammar, Head-driven phrase structure grammar (HPSG) focuses more on lexicon details and defines phrase structures by a set of rules that describe how different types of heads combine with their complements and specifiers. A central innovation in HPSG is its use of feature structures to represent syntactic, semantic, and morphological information. Attribute-value matrices encode intricate linguistic information as features. HPSG also uses unification, a process in which different feature structures are merged under constraint that all relevant syntactic and semantic information is consistently maintained.
[4] Combinatory Category Grammar (CCG) is an efficient parable and linguistically expressive grammar fomalism, which generates consistency-based structures (as opposed to a dependency grammar). It is not like traditional phrase structure grammar using rules, instead, CCG assumes every word contains information of how to be combined with other words. Each word is defined by a categroy in CCG, which specifies its grammar type and what kind of adjacent words can be used to integrate a larger structure. When using CCG provided in https://github.com/chrzyki/candc, an example of sentence 'The government plans to raise the tax.' is as follows.

```
(<T S[dcl] 1 2> (<T S[dcl] 0 2> (<T NP 0 1> (<T N 0 2> (<L N/N NP[nb]/N NP[nb]/N The N/N>) (<L N N/N N/N government N>))) (<T S[dcl]\NP 1 2> (<l
```

Specification:

`<T Category start end>` nonterminal node (tree), category of combination, start and end position.
`<L ....>` terminal node (leaf)
`S[dcl]` a decalritive sentence
`S[to]` infinite "to do sth."
`S[b]` verb phrase
`/` forward-looking A/B B->A `The|NP/N  government|N  →  NP` , which means that 'The' requires a Noun on the right to become a NP.
`\` backword-looking A\B A->B `plans|(S[dcl]\NP)/NP  income tax|NP  →  S[dcl]\N` , which means that 'plans' needs a NP on the left to form a declaritive sentence.
By applying the functions, words can be combined to construct a larger structre. However, the result is complicated due to the format.

(b)
I experiment Denpendency grammer with Chinese and find out that it still only shows the relationship between pairs of preterminals. However, in the context of Chinese, dependent grammar sometimes assigns two words with only one preterminal, which is contrast with the version of parsing English sentences. For example, it assigns VERB to '觉得', which could be translated as 'think'. This is rational because '觉得' itself is a disyllabic verb. However, this could cause misleading. For instance, I simply test it with a question sentence 'do you think it is proper of you doing this?', which is translated as '你觉得你这么做好吗'. It assigns '做好' with VERB, but '做好' is actually not a disullabic verb. Instead, the sentence should be seperated syntactically as '你觉得你这么做' (do you think that you doing this) and '好吗' (is proper), which mean '做好' is VERB+adj instead of a verb like '觉得'. What's more, when I test with '你最近怎么样' (how are you recently), where '最近' (recently) is ADV, '怎么样' (semantically similar to 'how') is ADJP, and an auxiliary verb is omitted orally. However, parser assign '最近' as NOUN and '怎么样' as ADV, which are not correct. Based on these cases, the parses are not very accurate as it has done on English sentences.

Q3

(a)

Base on the recognizer, each grammar line is read into a `Rule` whose weight is the negative log-probability, so we can accumulate costs by addition. An Earley item still records `(rule, dot_position, start_position)` , but the column agenda now maintains dictionaries for the current best weight and the backpointer of every item. Predict enqueues a new item with the rule's weight and a sentinel backpointer that marks the start of the derivation. Scan propagates the parent weight unchanged because terminals have no additional cost, while attach merges the parent prefix weight with the completed child's lowest score before advancing the dot. Whenever the chart finishes, we sweep the last column for complete ROOT items that span the whole sentence, the backpointer chain of the lightest one is then walked recursively to print the minimum weight tree.

(b)

`Agenda.push_or_move` keeps a single entry per rule. When an item appears for the first time it is stored, indexed under its pending next symbol, and associated with its weight/backpointer pair. If later processing discovers the same dotted rule with a smaller weight, this method overwrites the weight and backpointer, and if the item had already been popped, un-pops it by swapping it just before `_next` , ensuring the improved item will be processed again. In this way the chart performs exact lowest score search without producing duplicate queue entries. Since there are only `n+1` columns and each item is uniquely determined by its start position, dot position, and grammar rule, the total number of stored items is $O(n^2)$, matching the usual Earley space bound. Every push (including the duplicate check) is $O(1)$: it uses a single hash lookup in `_index` , an optional append to `_items` , and a constant number of dictionary updates when registering the item in `_waiting` . That constant-time agenda maintenance is what lets the predict/scan/attach schedule stay within the standard $O(n^3)$runtime—any slower push would blow up that bound because each completed item can trigger $O(n)$ further operations in $O(n)$columns.

Reprocessing (extra credit; reading section B.2)

I followed B.2 "move" reprocessing: when `push_or_move` finds a lower-weight duplicate, it overwrites the item's stored weight/backpointer and swaps the item back into the unpopped region of the column in O(1) so it will be processed again. Since all rule weights are −log probabilities (≥0) and scans add 0, weights are monotone, hence every improvement strictly lowers the value, so this cannot loop.

Let V be the number of distinct items, which is $O(n^2)$ for a fixed grammar, and let E be the number of attach "uses," which is $\Theta(n^3)$. Each time Z improves, we redo its $O(\text{outdegree}(Z))$ attachments. In highly ambiguous grammars, Z could theoretically improve many times (once per distinct derivation path to Z). More precisely, the total runtime is

$$\Theta(n^3) \; + \sum_{Z} (\#\text{improvements of } Z) \cdot \text{outdegree}(Z).$$

To avoid reprocessing, we could use a Global best-first agenda (Dijkstra). Replace the per-column FIFO with one global min-heap keyed by item weight. With non-negative edge costs (rule ≥0, attach adds a completed child's cost ≥0, scan 0), once an item pops from the heap, its weight is final and can never be improved later. Hence no reprocessing is needed. The complexity is $O((V+E)\log V) = O(n^3 \log n)$ with a binary heap.

Q4

I profiled various speedups on `wallstreet.gr` / `wallstreet.sen` . The main hotspots were repeated predictions and huge numbers of dotted-rule states. I addressed those issues in stages and the final submitted `parse2.py` combines the most effective optimizations:

- **Prediction memoization (E.1).** The first attempt maintains a column-level cache so each `(nonterminal, position)` pair is expanded only once. This prevents every customer from re-reading the same rule list. Since later I truned to Trie for grammars so this is not used anymore in the final submission.
- **Vocabulary specialization (E.2).** For the sentence being parsed I collect all token ids and restrict predictions to nonterminals that can reach at least one of those terminals, plus the start symbol. The grammar tracks reachable terminals per nonterminal while it is building its tries. Items whose outgoing symbols fall outside this allow-set are never scheduled, which prunes large sections of the chart.
- **Trie-based Earley items (E.4).** Each left-hand side owns a trie that merges all right-hand-side prefixes and stores the rule weight at terminal nodes. An item is now `(lhs id, trie state, start_position)` , so a single state represents every dotted rule that shares the same prefix. When an item pops, I enumerate all outgoing nonterminal for prediction, follow any matching terminal to scan, and when a node is final I add the cached rule weight while attaching to waiting parents. The agenda indexes each item under all of its future nonterminals, which keeps attach lookups constant time.
- **Integerized symbols.** I reused the `Integerizer` from previous homeworks to map every terminal and nonterminal to an id as soon as the grammar loads. The sentence tokens are cached as ids, trie edges are stored as integer-labelled transitions, and the agenda's waiting dictionary is keyed by integer nonterminals. This removes the repeated hashing of Python strings inside all hot loops.

Together these changes shrink both the number of states we explore and the per-state processing cost, which is reflected in the expriment results below.

Experiment results

Wall Street parsing times (Using PyPy on M3 Max):

parser used	time taken (s)
baseline probabilistic parser (`parse.py`)	902.43
+ E.1 batch duplicate check	279.82
+ E.1 and E.2 vocabulary specialization	265.60
+ E.2 and E.4 trie items	75.69
+ E.2, E.4, and integerization (final `parse2.py`)	71.97

The final submission is over 12× faster than the unoptimized parser on this benchmark while still returning identical best parses.

Comment on Parses

1. Interesting points

1. There are multiple tags with specific functions in the tree. For example, ADJP-PRD represents predictive adjective phrase, S-ADV represents adverbial clause, and S-NOM is nominalized clause. This style of parser tends to specify the syntax structure of the sentence instead of just NP, VP, and PP, which is more detailed and useful for understanding. The complex sentences are decomposed into deeply nested clauses such as SBAR (said the administration considered ...), S-ADV (... companies , projected to have ...), and S-NOM (critisized for issuing ...). This showcases that the parser prefers a fine-grained, function-oriented style rather than flatter analysis.
2. Punctuation marks are explicitly represented as their own nodes instead of affiliate constituents, which is rare in parsers.

2. Correct and wrong things parser does

1. The parser handled many difficult constructions correctly, such as predicative adjectives ("John is happy", where 'happy' is assigned with ADJP-PRD), embedded complecx clauses (e.g., in "a senior intelligence official said the administration considered...", multiple clauses are correctly parsed), and participial adverbials ("caught off guard" is tagged as S-ADV).
2. However, it also made some mistakes: proper names were sometimes split into multiple nodes (e.g., "Ford Motor Co." is tagged as 3 consecutive NPPs, which should be regarded as a whole), complement structures like "data show that pay was flat" were parsed in a non-standard way (where 'pay' should be a noun yet was tagged as a verb), long sentences with multiple modifiers were somewhat cluttered ("running the combined ... Wall Street analysts."), and quotation marks were attached as separate punctuation nodes rather than integrated into the sentence structure (the last sentence in the example), which is quite weird.