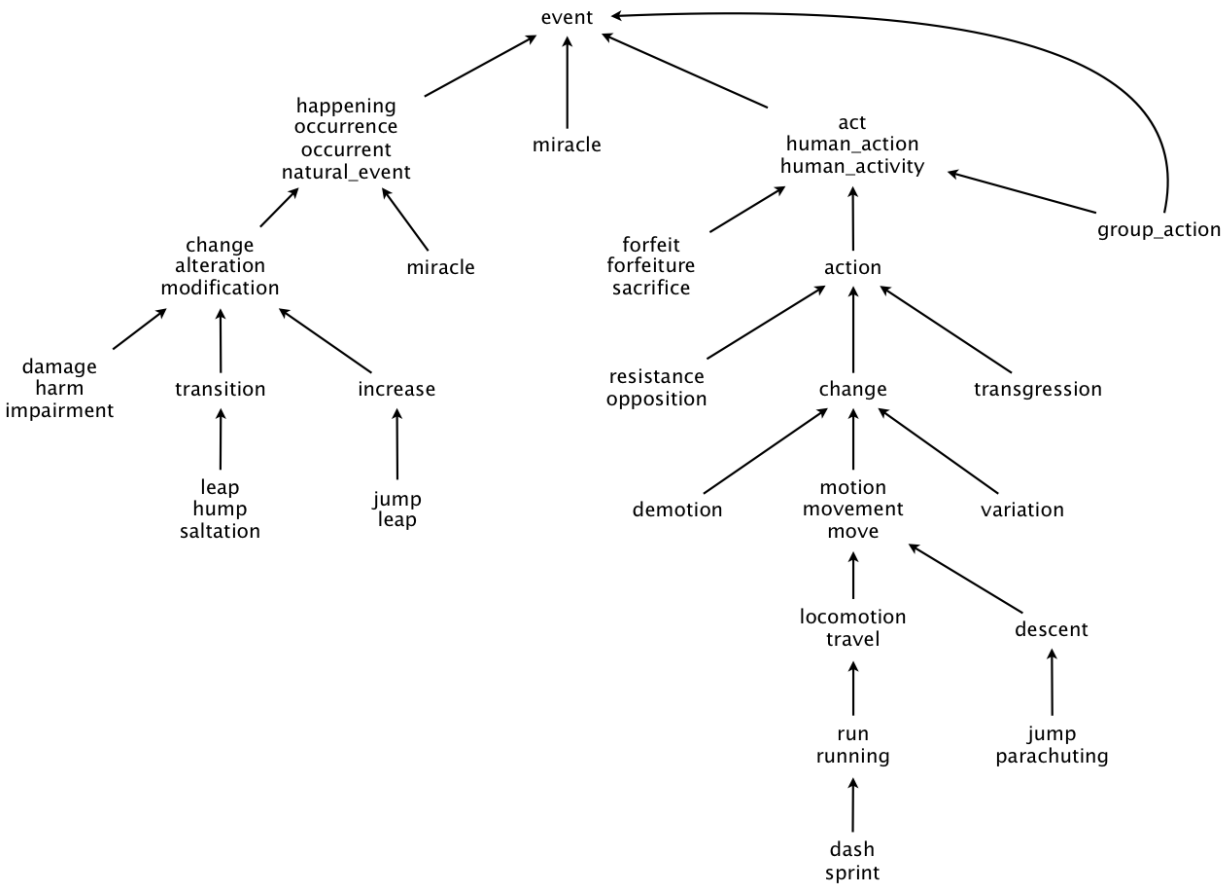
Homework # 03 *Due: 5 Dec 2025*

WordNet

Introduction

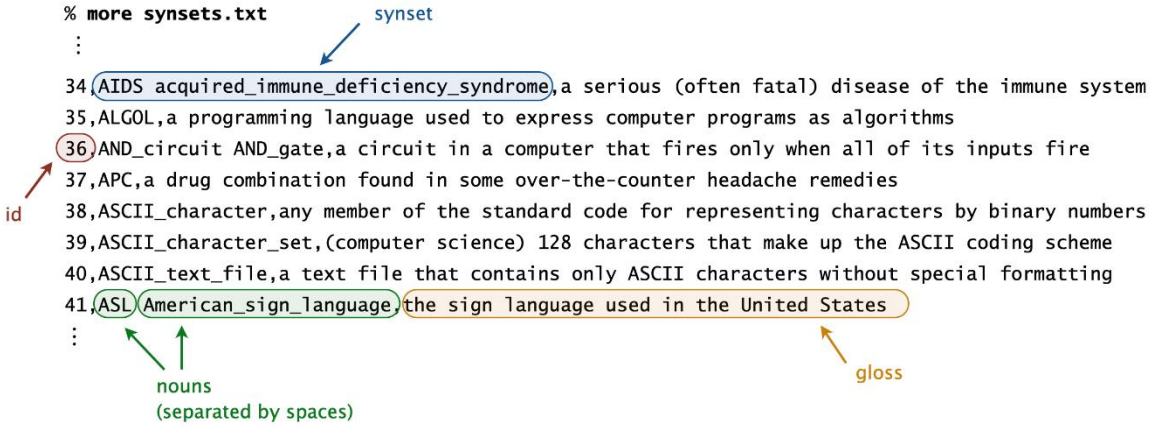
WordNet is a semantic lexicon for the English language that computational linguists and cognitive scientists use extensively. For example, WordNet was a key component in IBM's Jeopardy-playing Watson computer system. WordNet groups words into sets of synonyms called synsets. For example, { AND circuit, AND gate } is a synset that represent a logical gate that fires only when all of its inputs fire. WordNet also describes semantic relationships between synsets. One such relationship is the is-a relationship, which connects a hyponym (more specific synset) to a hypernym (more general synset). For example, the synset { gate, logic gate } is a hypernym of { AND circuit, AND gate } because an AND gate is a kind of logic gate.

The WordNet digraph. Your first task is to build the WordNet digraph: each vertex v is an integer that represents a synset, and each directed edge v→w represents that w is a hypernym of v. The WordNet digraph is a rooted DAG: it is acyclic and has one vertex—the root—that is an ancestor of every other vertex. However, it is not necessarily a tree because a synset can have more than one hypernym. Here is a small subgraph of the WordNet digraph:



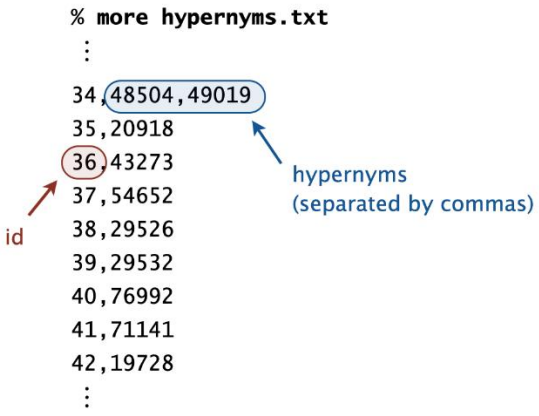
The WordNet input file formats. We now describe the two data files that you will use to create the WordNet digraph. The files are in *comma-separated values* (CSV) format: each line contains a sequence of fields, separated by commas.

- *List of synsets.* The file synsets.txt contains all noun synsets in WordNet, one per line. Line *i* of the file (counting from 0) contains the information for synset *i*. The first field is the *synset id*, which is always the integer *i*; the second field is the synonym set (or *synset*); and the third field is its dictionary definition (or *gloss*), which is not relevant to this assignment (and so you can ignore it).



For example, line 36 means that the synset { AND_circuit, AND_gate } has an id number of 36 and its gloss is a circuit in a computer that fires only when all of its inputs fire. The individual nouns that constitute a synset are separated by spaces. If a noun contains more than one word, the underscore character connects the words (and not the space character).

- *List of hypernyms.* The file hypernyms.txt contains the hypernym relationships. Line *i* of the file (counting from 0) contains the hypernyms of synset *i*. The first field is the synset id, which is always the integer *i*; subsequent fields are the id numbers of the synset's hypernyms.



For example, line 36 means that synset 36 (AND_circuit AND_Gate) has 43273 (gate logic_gate) as its only hypernym. Line 34 means that synset 34 (AIDS acquired_immune_deficiency_syndrome) has two hypernyms: 48504 (immunodeficiency) and 49019 (infectious_disease).

WordNet data type. Implement an immutable data type WordNet with the following API:

```
class WordNet {
public:
    // Constructor takes two input file names (synsets and hypernyms)
    WordNet(const std::string& synsetsFile,
            const std::string& hypernymsFile);

    // returns all WordNet nouns
    const std::unordered_set<std::string>& nouns() const;

    // is the word a WordNet noun?
    bool isNoun(const std::string& word) const;

    // a synset (i.e., the synonym set) that is a shortest common ancestor of noun1 and noun2
    std::string sca(const std::string& noun1, const std::string& noun2) const;

    // distance between noun1 and noun2
    int distance(const std::string& noun1, const std::string& noun2) const;
};
```

Corner cases:

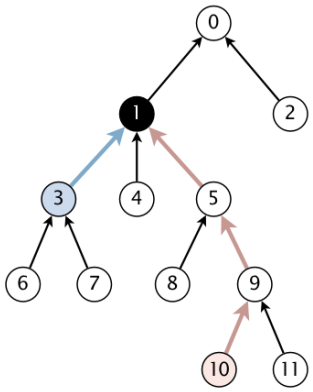
- If the constructor argument is "" (empty) or if the file cannot be opened, throw `std::invalid_argument`.
- If either noun1 or noun2 is not a WordNet noun when calling `sca()` or `distance()`, throw `std::invalid_argument`.

You may assume that the input files are in the specified format and that the underlying digraph is a rooted DAG.

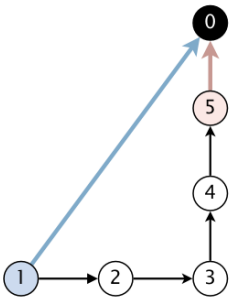
Performance requirements. Your implementation must achieve the following performance requirements. In the requirements below, assume that the number of characters in a noun or synset is O(1).

- Your data type must use space linear in the input size (size of synsets and hypernyms files).
- The constructor must take time linearithmic (or better) in the input size.
- The method `isNoun()` must run in time logarithmic (or better) in the number of nouns.
- The methods `distance()` and `sca()` must make exactly one call to the `lengthSubset()` and `ancestorSubset()` methods in ShortestCommonAncestor, respectively.

Shortest common ancestor. An *ancestral path* between two vertices *v* and *w* in a rooted DAG is a directed path from *v* to a common ancestor *x*, together with a directed path from *w* to the same ancestor *x*. A *shortest ancestral path* is an ancestral path of minimum total length. We refer to the common ancestor in a shortest ancestral path as a *shortest common ancestor*. Note that a shortest common ancestor always exists because the root is an ancestor of every vertex. Note also that an ancestral path is a path, but not a directed path.

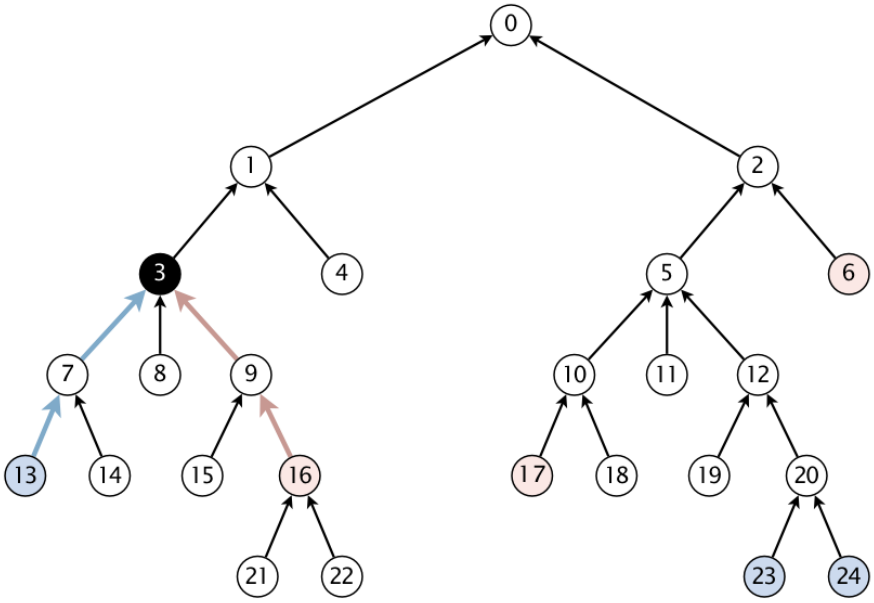


v = 3, w = 10
shortest ancestral path: 3–1–5–9–10
associated length: 4
shortest common ancestor: 1



v = 1, w = 5
ancestral path: 1–2–3–4–5
shortest ancestral path: 1–0–5
associated length: 2
shortest common ancestor: 0

We generalize the notion of shortest common ancestor to *subsets* of vertices. A shortest ancestral path of two subsets of vertices *A* and *B* is a shortest ancestral path among all pairs of vertices *v* and *w*, with *v* in *A* and *w* in *B*. As an example, the following figure (digraph25.txt) identifies several (but not all) ancestral paths between the red and blue vertices, including the shortest one.



A = { 13, 23, 24 }, B = { 6, 16, 17 }
ancestral path: 13–7–3–1–0–2–6
ancestral path: 23–20–12–5–10–17
ancestral path: 23–20–12–5–2–6

shortest ancestral path: 13–7–3–9–16
associated length: 4
shortest common ancestor: 3

Shortest common ancestor data type. Implement an immutable data type ShortestCommonAncestor with the following API:

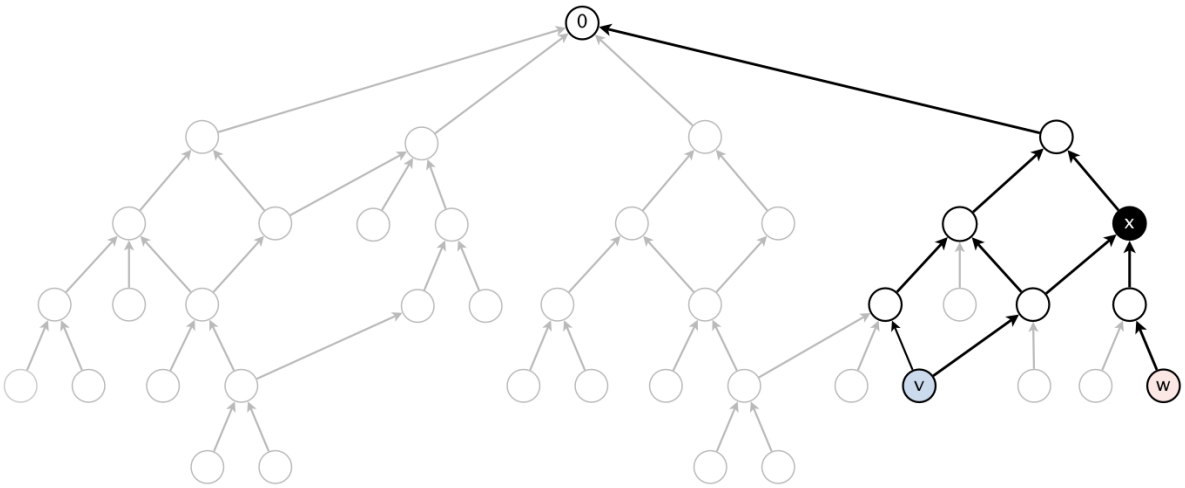
```
class ShortestCommonAncestor {
public:
    // Constructor takes a rooted DAG
    ShortestCommonAncestor(const Digraph& G);

    // length of shortest ancestral path between v and w
    int length(int v, int w) const;

    // a shortest common ancestor of v and w
    int ancestor(int v, int w) const;

    // length of shortest ancestral path between subsets A and B
    int lengthSubset(const std::vector<int>& subsetA,
                     const std::vector<int>& subsetB) const;

    // a shortest common ancestor of subsets A and B
    int ancestorSubset(const std::vector<int>& subsetA,
                       const std::vector<int>& subsetB) const;
};
```

Corner cases:

- The constructor should throw `std::invalid_argument` if G is not a rooted DAG (i.e., if it has a cycle or more than one root or unreachable vertices).
- All methods should throw `std::invalid_argument` if any argument is invalid (e.g., indices out of range, empty subsets, or subset vectors containing invalid/duplicate vertices?).

Basic performance requirements. Your implementation must achieve the following worst-case performance requirements, where *E* and *V* are the number of edges and vertices in the digraph, respectively.

- Your data type must use O(E+V) space.
- All methods (except those mentioned in the next point) and the constructor must take O(E+V) time.
- The methods `length()`, `lengthSubset()`, `ancestor()`, and `ancestorSubset()` must take time proportional to the number of vertices and edges *reachable* from the argument vertices (or better), For example, to compute the shortest common ancestor of *v* and *w* in the following digraph, your algorithm can examine only the highlighted vertices and edges; it cannot initialize any vertex-indexed arrays.



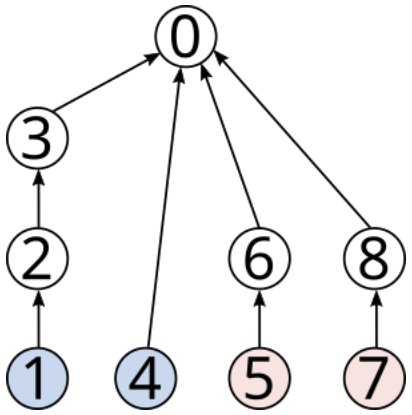
Unit testing. In your `main()` method you will implement a test that creates a specific digraph for which you know the `lengthSubset()` of. Before describing the unit test, we will remind you of a definition.

A *directed path of length* ℓ is a sequence of ℓ directed edges that connect ℓ+1 vertices. The *starting vertex* of a path is the vertex from which the path starts, i.e., the sole vertex with one outgoing and no incoming path edges. The *ending vertex* of a path is the vertex from which the path ends, i.e., the sole vertex with one incoming and no outgoing path edges. (All other vertices have one outgoing and one incoming edge each.)

The unit test you must implement is the following:

- Get two integers n and m as a command-line arguments and create two arrays of n integers each, pathsA and pathsB. Fill each array with uniform random integers between 1 and m.
- The rooted DAG you shall create should be a graph given by the union of 2n directed paths all with the same ending vertex, which is thus the root vertex (it might simplify your implementation to designate vertex 0 as the root vertex). The lengths of these paths shall be given by the values in pathsA and pathsB.
- Let SA and SB be the sums of pathsA and pathsB, respectively. Create a Digraph with SA+SB+1 vertices, whose edges form 2n edge-disjoint paths with lengths given by pathsA and pathsB (see below for an example).
- Create a ShortestCommonAncestor object with this graph and run lengthSubset() with subsetA given by the starting vertices corresponding to the directed paths from pathsA, and subsetB given by the starting vertices corresponding to the directed paths from pathsB.
- Note that the expected result of the above lengthSubset() run should be exactly the sum of the minimum integer in pathsA and the minimum integer in pathsB. Thus, check that the output of lengthSubset() matches this sum and print a message (e.g. "Error!") if not.

As an example, suppose n=2, m=3, pathsA = {3, 1}, and pathsB = {2, 2}. Then the following image represents this instance, with the colored vertices representing the two subsets.



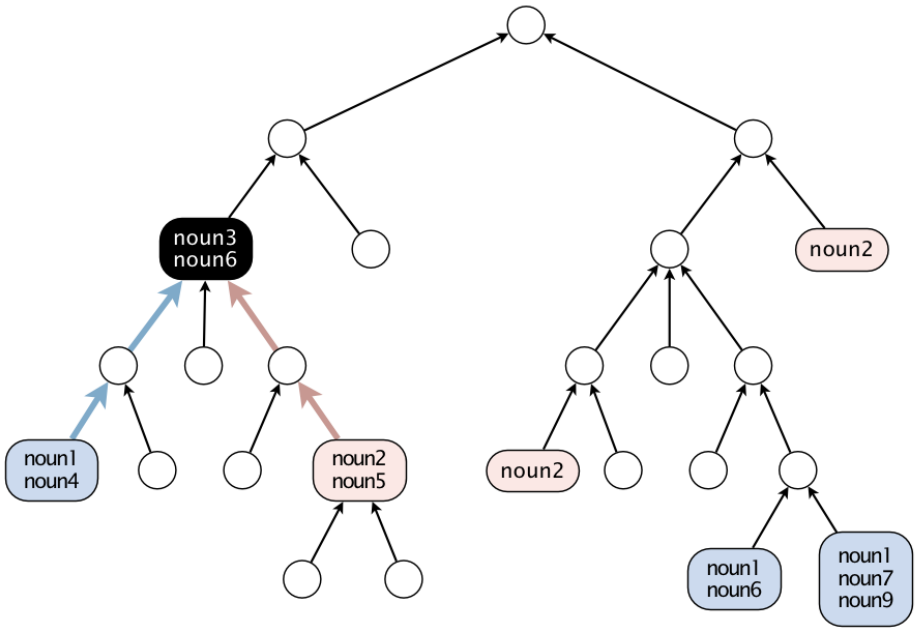
Note that the number of vertices and edges in this rooted DAG is $\Theta(nm)$, so a correct implementation should take no more than a few seconds when n and m are 1,000.

Measuring the semantic relatedness of two nouns. Semantic relatedness refers to the degree to which two concepts are related. Measuring semantic relatedness is a challenging problem. For example, you consider *George W. Bush* and *John F. Kennedy* (two U.S. presidents) to be more closely related than *George W. Bush* and *chimpanzee* (two primates). It might not be clear whether *George W. Bush* and *Eric Arthur Blair* are more related than two arbitrary people. However, both *George W. Bush* and *Eric Arthur Blair* (a.k.a. George Orwell) are famous communicators and, therefore, closely related.

We define the semantic relatedness of two WordNet nouns x and y as follows:

- A = set of synsets in which x appears
- B = set of synsets in which y appears
- *distance*(x, y) = length of shortest ancestral path of subsets A and B
- *sca*(x, y) = a shortest common ancestor of subsets A and B

This is the notion of distance that you will use to implement the `distance()` and `sca()` methods in the WordNet data type.



Outcast detection. Given a list of WordNet nouns x_1, x_2, ..., x_n, which noun is the least related to the others? To identify *an outcast*, compute the sum of the distances between each noun and every other one:

$$d_i \;=\; distance(x_i, x_1) \;+\; distance(x_i, x_2) \;+\; \ldots \;+\; distance(x_i, x_n)$$

and return a noun x_t for which d_t is maximum. Note that $distance(x_i, x_i) = 0$, so it will not contribute to the sum.

Implement an immutable data type Outcast with the following API:

```
class Outcast {
public:
    // Constructor takes a WordNet object
    Outcast(const WordNet& wn);

    // given a vector of WordNet nouns, return the outcast (least related noun)
    std::string outcast(const std::vector<std::string>& nouns) const;
};
```

Corner cases. Assume that the argument to `outcast()` contains only valid WordNet nouns and that it contains at least two such nouns.

Deliverables

Submit the following source files:

- `WordNet.cpp`
- `ShortestCommonAncestor.cpp`
- `Outcast.cpp`
- A `main.cpp` file for each of the classes where required (e.g., for testing `ShortestCommonAncestor` and demonstration code for WordNet/Outcast).
- `readme.txt` (documentation of how to compile, run, any known issues)

You may **not** use any library functions except those in the C++ Standard Library (e.g., `<vector>`, `<string>`, `<unordered_map>`, `<unordered_set>`, `<queue>`, `<stack>`, `<stdexcept>`, `<fstream>`, etc.). You may **not** use external graph libraries for digraph processing (you must implement any required algorithms yourself).

Grading

- `WordNet.cpp`: 12 points
- `ShortestCommonAncestor.cpp`: 16 points
- `Outcast.cpp`: 6 points
- `readme.txt`: 6 points
- Total: 40 points

Notes & Tips

- Use zero-based synset IDs (as in the input files).
- Use `std::unordered_map<std::string, std::vector<int>>` to map each noun → list of synset-IDs containing it.
- Use `std::vector<std::string>` to map synset-ID → synset string.
- Represent the digraph with adjacency lists: `std::vector<std::vector<int>> adj;`
- To check for DAG & rooted-ness: ensure no cycles (via DFS algorithm) and exactly one vertex with out-degree 0 (the root).
- Implement BFS/DFS from subsets in `ShortestCommonAncestor` to find shortest ancestral path.
- For performance: reuse visited arrays, limit per-call resets, avoid full graph scans when possible.
- Provide clear error handling: check for invalid inputs, etc.
- Document your code, include comments, follow a consistent naming/style convention.
- In `readme.txt`, describe how to compile (e.g., `g++ -std=c++17 *.cpp -o wordnet`) and how to run your tests.

Acknowledgement

This assignment is adopted from
https://www.cs.princeton.edu/courses/archive/fall25/cos226/assignments/wordnet/specification.php
The following FAQs are adopted from
https://www.cs.princeton.edu/courses/archive/fall25/cos226/assignments/wordnet/checklist.php

Frequently Asked Questions (FAQ)

For `WordNet`

- **Can I read the synsets or hypernyms file twice?**
  No. File I/O is relatively expensive; you should read each file **only once**, and then organize the data into appropriate data structures for look-ups.
- **Any advice on how to read and parse the synset and hypernym files?**
  Read each line at a time (e.g., using `std::getline` on an `std::ifstream`), then split the line into fields (e.g., using `std::istringstream` or `std::string::find/substring`). Convert numeric fields (IDs) into `int` via `std::stoi`.
- **Which data structures should I use to store the synsets/nouns/hypernyms?**
  That is up to you. You must choose data structures that allow you to satisfy the specified performance bounds (e.g., constant or logarithmic look-up for nouns).
- **How do I map a noun to the synset IDs that contain it?**
  You can use a map/dictionary such as `std::unordered_map<std::string, std::vector<int>>`, mapping each noun → list of synset IDs that contain that noun.
- **What should `nouns()` return: duplicates or no duplicates?**
  `nouns()` should return each distinct noun exactly once (i.e., no duplicates) since you're returning the *set* of all nouns.
- **Does `nouns()` need to return nouns in alphabetical order?**
  No. The API does not require alphabetical ordering; you are free to return in any order.
- **Do I need to store the glosses (dictionary definitions) from the synsets file?**
  No—those are not used in this assignment. You may ignore them.
- **How large is the WordNet DAG?**
  For the standard data set, there are about 83,127 synsets, 85,441 hypernym pairs, and 120,119 distinct nouns. But you **must not** hard-code those numbers—your program must work for *any* valid synsets/hypernyms input.
- **Can a synset contain more than one noun?**
  Yes—synsets are sets of one or more nouns.
- **Can a noun contain underscore characters?**
  Yes—for example `American_sign_language` is one noun.
- **Can a noun contain spaces?**
  No. Nouns will not contain space characters (spaces are used to separate multiple nouns within a synset).
- **Can a noun appear in more than one synset?**
  Yes, absolutely. A noun may have multiple meanings and thus appear in multiple synsets.
- **Can a synset consist of exactly one noun?**
  Yes—and you may have several different synsets with the *same* noun (corresponding to different meanings).

For `ShortestCommonAncestor` (SCA)

- **Can I use my own graph class?**
  If you use a `Digraph` class, that's fine—but you must ensure the constructor of your SCA type takes a graph object that matches the expected semantics (rooted DAG check etc).
- **How can I make `ShortestCommonAncestor` (or SCA) *immutable*?**
  You should store a copy of the graph in the object (do a defensive copy in the constructor) so that subsequent modifications of the original graph will not affect the SCA object's internal state.
- **Is a vertex considered an ancestor of itself?**
  Yes. A node is considered its own ancestor in this context.

- **What should `ancestor()` return if there is a tie for the shortest common ancestor?**
  The API does *not* specify a particular tie-breaking rule, so you are free to return *any* one of the valid shortest common ancestors.
- **I understand how to compute `length(int v, int w)` in O(E+V) time, but my `lengthSubset(...)` is taking O(a × b × (E+V)). How can I improve it to O(E+V)?**
  The key is to perform a *multi-source breadth-first search (BFS)* from all vertices in subset A simultaneously and/or all vertices in subset B simultaneously, rather than nested loops for each pair. That way you get something like O(E+V) regardless of the sizes of the subsets.
- **Should I construct a new `ShortestCommonAncestor` object for each call to `distance()` or `ancestor()`?**
  No. Create one SCA object (one graph + one object) and reuse it for multiple queries.
- **How can I determine whether a digraph is a *rooted DAG*?**
  Break it into two checks:
  1. Is the digraph acyclic? (You can check that via DFS or topological sort.)
  2. Does it have exactly one root (one vertex with zero out-degree, to which all vertices have a path)?
- **Do I need to re-implement DFS or BFS from scratch?**
  You may reuse existing standard BFS/DFS implementations (or write your own) as long as you adhere to the performance requirements and clearly document what you reuse.

For `Outcast`

- **What should `outcast()` return if there is a tie (two nouns have the same maximum "outcast-score")?**
  The API does not specify a tie-breaking rule, so you may return any one of the tied nouns.
- **My algorithm computes the distance between *every* pair of nouns. Is that okay?**
  Yes. That is perfectly acceptable (though possibly inefficient for very large sets).

Input/Output, Testing & Other Notes

- **Are we required to handle only the standard WordNet data set?**
  No. Your solution must work for *any* valid synsets/hypernyms input files of the correct format, not just the WordNet data.
- **What about performance?**
  You must meet the performance bounds specified in the assignment (space linear in input size; constructor in about O(N) or O(N log N); methods for queries in about O(E+V) or better).
- **Can I submit any helper test inputs or corner cases?**
  Yes—and doing so may even earn bonus credit (especially if you find interesting corner-cases). Document clearly in your `readme.txt`.
- **What should I include in `readme.txt`?**
  Explain how to compile and run your code (e.g., `g++ -std=c++17 *.cpp -o wordnet`), list any known issues or limitations, explain how you tested (especially corner cases), and any design decisions (data structures chosen, why). See the format available in the shared file.

Additional FAQs can be found here
https://www.cs.princeton.edu/courses/archive/fall25/cos226/assignments/wordnet/checklist.php