16. List Processing Case Study

CPSC 120: Introduction to Programming

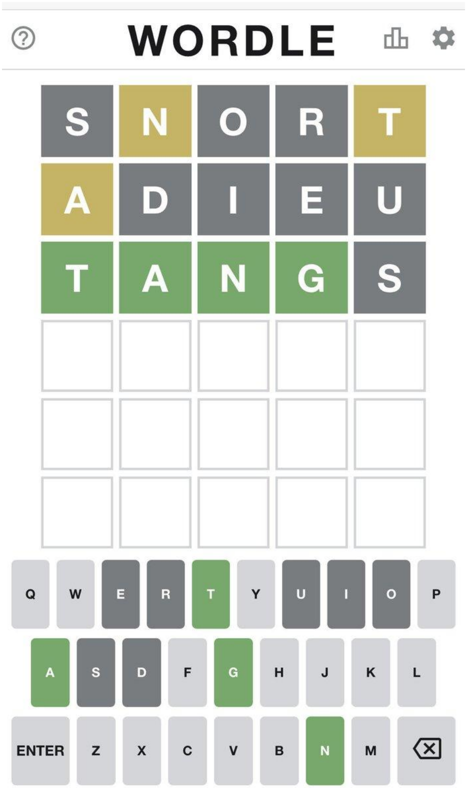
Kevin A. Wortman ~ CSU Fullerton

Agenda

1. Reminder: Midterm 2 Tomorrow
2. Domain: Wordle Hint
3. Review: Designing Functions
4. Review: List Processing Patterns
5. Live Coding

1. Domain: Wordle Hint

Wordle


WORDLE
S N O R T
A D I E U
T A N G S
Q W E R T Y U I O P
A S D F G H J K L
ENTER Z X C V B N M

Wordle Clue

- **Clue:** a string containing
  - characters for known letters
  - ? for unknown letters
- Examples
  - “tang?”
  - “vam????”
  - “??oon”

words.txt

- Unix: /usr/share/dict has **dictionary files**
- **dictionary file:** contains valid words, one word per line
- Start of /usr/share/dict/words:

```
A
A's
AMD
AMD's
AOL
AOL's
AWS
AWS's
Aachen
Aachen's
Aaliyah
Aaliyah's
Aaron
Aaron's
...
```

Word Frequencies

- **frequency:** how many times something occurs
- Ilya Semenov's frequency data from Wikipedia: https://github.com/IlyaSemenov/wikipedia-word-frequency
- Start of enwiki-2022-08-29.txt:

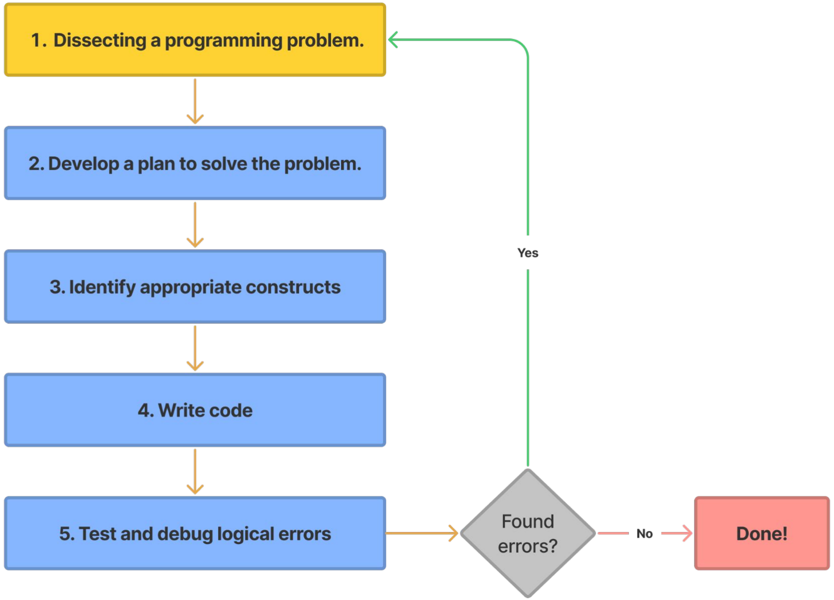
```
the 183212978
of 86859699
in 75290639
and 74708386
a 53698262
to 52250362
was 32540285
is 23812199
on 21691194
for 21634075
as 21126503
with 18605836
...
```

Wordle Hints

- INPUT: a clue string, ex. “tang?”
- OUTPUT: a list of English words that match the clue
- (bonus): indicate which match has the highest frequency

2. Review: Designing Functions

Steps for Solving a Programming Problem



1. Dissect the Problem

- **Understand the problem:** read three times, take notes
- **Identify inputs:** what are the parameters (inputs)
- **Identify outputs:**
  a. What is the return type? Or is it void?
  b. Does the function have side effects? What are they?
- **Identify test cases:** what happens for...
  a. ordinary arguments
  b. unusual arguments

2. Develop a Plan

1. Express the solution in an understandable form
   a. **Write outline** of function: prototype and TODO comments in the body
2. Test your solution
   a. Trace outline; decide whether it has logic errors
3. Revise solution
   a. If there is a logic error, revise outline

```cpp
std::string prompt_string(std::string query) {
 // TODO: declare variable for result
 // TODO: print prompt using query parameter
 // TODO: use cin to read input
 // TODO: return result variable
}
```

3. Identify Appropriate Constructs

- Identify code constructs for each TODO comment
- Local variables
- Statements inside body: assignment, if, loops, etc.
- (value-returning function) Return statement(s)

4. Write Code

- Replace each TODO comment with working statements
- Work one at a time
- Small bites

5. Test and Debug Errors

- As usual, test your program
- Debug
  - Compile errors
  - Logic errors
  - Runtime errors

3. Review: List Processing Patterns

Algorithm: Build a Vector

OUTPUT: a vector OUT of elements

1. Declare OUT as an empty vector
2. Loop for each element:
   a. Create a new element
   b. Add the element to the back of OUT (push_back)

```
std::vector<int> years; // 20th century years
for (int i = 1900; i < 2000; ++i) {
  years.push_back(i);
}

 std::vector<std::string> arguments(argv, argv +
argc);

// just the inputs; skip the command name
std::vector<std::string> inputs;
for (int i = 1; i < arguments.size(); ++i) {
  inputs.push_back(arguments.at(i));
}
```

```
$ ./blackjack J 6 A

arguments: "./blackjack" "J" "6" "A"
inputs: "J" "6" "A"
```

Algorithm: Build Vector From File

(variation on **Build a Vector**)

INPUT: a filename containing data elements
OUTPUT: a vector OUT containing the elements from the file

1. Open input file (`ifstream`)
2. Declare OUT as an empty vector
3. while file is good:
   a. Read one data element
   b. if no I/O error: add element to back of OUT (push_back)

```
std::vector<std::string> words;
std::ifstream file("words.txt");
while (true) {
  std::string one_word;
  file >> one_word;
  if (!file.good()) {
    break;
  }
  words.push_back(one_word);
}
```

Algorithm: Filter

INPUT: a vector IN of elements that may match
OUTPUT: a vector OUT containing only the matching elements

1. Declare OUT as an empty vector
2. For each element x of IN:
    a. if x is a match:
        i. Add x to back of OUT (push_back)

```
std::vector<std::string> arguments(argv, argv + argc);

std::vector<std::string> cards;
for (int i = 1; i < arguments.size(); ++i) {
  cards.push_back(arguments.at(i));
}

std::vector<std::string> aces;
for (const std::string& card : cards) {
  if (card == "A") {
    aces.push_back(card);
  }
}
```

```
$ ./blackjack A 3 K A

arguments: “./blackjack” “A” “3” “K” “A”
cards: “A” “3” “K” “A”
aces: “A” “A”
```

Review: Linear Search

- Linear search: algorithm for finding an element, which may not exist, in a container
  - Brute force password cracking
  - Ray tracing computer animation
  - Basis of algorithm patterns: exhaustive search, greedy pattern, Monte Carlo method
- Check each element in order
- If the current one is what we want, stop (**success/found**)
  - Stop with `break`
- If we get to the end, and never found what we want, stop (**failure/not-found**)
- Two outcomes
  - **Found**: did find a match
  - **Not-found**: there is no match
- `bool` loop control variable for found/not-found

Review: Pattern: Linear Search

- `found`: `bool` variable remembers success/failure
- `match`: copy of matching element
  - only valid when `found` is `true`

bool found{false};
elt-type **match{***default-value***};**
for (const *elt-type***&** **element :** *container***) {**
 if (*elt-is-match-condition***) {**
  found = true;
  match = element;
  break;
 }
}
// use found and match

```
std::vector<int> values{5, 11, -2, 8};

// find a negative value
bool found{false};
int match{0};
for (const int& value : values) {
  if (value < 0) {
      found = true;
      match = value;
      break;
  }
}
if (found) {
  std::cout << match << " is a negative value\n";
} else {
  std::cout << "there are no negative values\n";
}
```

Algorithm: Unique Elements

INPUT: a vector IN, which may contain duplicates
OUTPUT: a vector OUT, where each element of IN appears only once

1. Declare OUT as an empty vector
2. for each element x of IN:
    a. Linear search for x in OUT
    b. If x is not found in OUT:
        i. Add x to OUT (`push_back`)

(There is a more efficient algorithm, see *CPSC 335 Algorithm Engineering*)

Example: Unique Elements

```
std::vector<std::string> signins;
// build vector...

std::vector<std::string> unique_names;
for (const std::string& signin : signins) {
  bool already_in_unique_names{false};
  for (const std::string& name : unique_names) {
    if (signin == name) {
      already_in_unique_names = true;
      break;
    }
  }
  if (!already_in_unique_names) {
    unique_names.push_back(signin);
  }
}
```

signins: “alice” “bob” “carlos” “alice” “bob”

unique_names: “alice” “bob” “carlos”

Algorithm: Common Elements

INPUT: vector L and vector R

OUTPUT: vector OUT contains every element that is in both L and R

1. Declare OUT as an empty vector
2. For each element x of L:
    a. Linear search for x in R
    b. If x is found in R:
        i. Add x to OUT (push_back)

(There is a more efficient algorithm, see *CPSC 335 Algorithm Engineering*)

Example: Common Elements

```
std::vector<int> years; // 20th century years
for (int i = 1900; i < 2000; ++i) {
  years.push_back(i);
}

std::vector<int> earthquake_years; // load from file...

std::vector<int> common_years;
for (int& x : years) {
  bool found{false};
  for (int& y : earthquake_years) {
    if (x == y) {
      found = true;
      break;
    }
  }
  if (found) {
    common_years.push_back(x);
  }
}
```

```
years: 1900, 1901, ..., 1999

earthquake_years: 1857, 1872, 1906,
1923, 1980, 1992, 2019

common_years: 1906, 1993, 1980, 1992
```

Algorithm: Transform

INPUT: a vector IN of elements
OUTPUT: a vector OUT of elements, containing a transformed version of each element of IN

1. Declare OUT as an empty vector
2. For each element x in IN:
    a. t = *transform* x
    b. Add t to back of OUT (`push_back`)

Example: Transform

```
std::vector<std::string> arguments(argv, argv + argc);

std::vector<std::string> cards;
for (int i = 1; i < arguments.size(); ++i) {
  cards.push_back(arguments.at(i));
}

std::vector<int> points;
for (const std::string& card : cards) {
  if (card == "A") {
    points.push_back(1);
  } else if ((card == "J") || (card == "Q") ||
             (card == "K")) {
    points.push_back(10);
  } else {
    points.push_back(std::stoi(card));
  }
}
```

```
$ ./blackjack “J” “7” “A” “K” “A”

arguments: “./blackjack” “J” “7” “A” “K”
“A”

cards: “J” “7” “A” “K” “A”

points: 10 7 1 10 1
```

4. Live Code Solution

Program Requirements

1. Input a **clue** as command line argument; validate
2. Read words from words.txt
3. Make list of English words that match the clue
4. Print the matches
5. (optional) Print out which match is most frequent

Example Input/Output

```
$ ./wordle tang?
Your clue matches:
tango
tangs
tangy
The most frequent is: tango

$ ./wordle ??cho
Your clue matches:
Tycho
macho
nacho
The most frequent is: macho
```