INF1008
Data Structures & Algorithms

Group 10

Contents

Question 1

1.1 Problem Statement

1.2 Requirements/Specification

1.3 User Guide

1.4 Structure/Design

1.5 Limitations

1.6 Testing

1.7 Listings

Question 2

2.1 Problem Statement

Write a program that reads in a sequence of characters, and determines whether its parentheses, braces, and curly braces are "balanced." Your program should read one line of input containing what is supposed to be a properly formed expression in algebra and tells whether it is in fact legal. The expression could have several sets of grouping symbols of various kinds, (), [], and {}. Your program needs to make sure that these grouping symbols match up properly. Analyse the efficiency of your implementation and provide a detailed discussion of its time and space complexity.

2.2 Requirements/Specification

Given any algebraic statement, (e.g. $-b \pm \left[\sqrt{\{b^2\} - (4)(a)(c)}\right] / 2(a)$), determine if the braces are balanced; That is, if the number of opening braces match the number of closing braces, and that the first closing brace matches with the last opening brace. The algorithm expects a properly formed expression in algebra as a string and outputs either `True` or `False`.

2.3 User Guide

To run the program, simply type `python main.py` in a terminal window. The program then prompts the user for an algebraic statement. If the statement is balanced, the program returns `True` and vice versa. No external libraries other than the standard `Python 3` libraries are required.

2.4 Structure/Design

The algorithm works by pushing opening brackets to a stack by looping through all bracket characters in the original statement. When encountering a closing bracket, it pops the last element of the stack and compares if they are complementary. If at any point the check fails, the algorithm returns `False` and ends the loop prematurely. At the end of the loop, the algorithm checks that the stack is empty. If it is, it returns `True` and `False` otherwise.

Algorithm 1: Bracket balance checker

```
Input: str statement

Function is_balanced(statement: str) is
    statement ← all brackets from statement;
    bracket_pairings ← {opening_bracket : closing_bracket};
    if len(statement)  mod 2 ≠ 0 then
        return False;
    end
    stack = [];
    foreach character in statement do
        if character is an opening bracket then
            stack.push(character);
        end
        else if bracket_pairing[stack.pop()] ≠ character then
            return False;
        end
    end
    return len(stack) == 0;
end
```

As the algorithm iterates through the input string only once, the time complexity is $O(n)$ for a given input string of length n. The opening brackets are iteratively pushed to and popped from a stack, so the space complexity is $O(n)$ as well.

Note:-

The time complexity of **Regular Expressions** and **Stack Operations** for insertion and deletion are known to be $O(n)$ and $O(1)$ respectively, so the time and space complexity of the algorithm remains at $O(n)$.

2.5 Limitations

While the algorithm determines perfectly if the brackets within any given algebraic statement are balanced, it does not check if the statement itself is a properly formed algebraic statement. Furthermore, it does not check that brackets on two sides of a given equality are balanced, as it only checks for bracket placement relative to other brackets in the entire input string (i.e. ([0]{=}2) will be evaluated as balanced).

2.6 Testing

Testing is handled by the `tests.py` file, which generates t input strings of up to length l, of which $n/2$ inputs are valid, and the other half are invalid. To generate valid inputs, the generator randomly selects an opening bracket or a closing bracket that matches the last opening bracket until reaching the halfway point of the string length, at which point it iteratively closes all the remaining open brackets. An example output of running the tests is shown in Figure 2.1.

Figure 2.1: Test output

```
Test 974:       ()): Passed (False)
Test 975:       [{{}}]: Passed (True)
Test 976:       [[][[: Passed (False)
Test 977:       (): Passed (True)
Test 978:       {{()}}: Passed (True)
Test 979:       {()}: Passed (True)
Test 980:       ({[{}]}): Passed (True)
Test 981:       {({(]({: Passed (False)
Test 982:       {: Passed (False)
Test 983:       ({[{}]}): Passed (True)
Test 984:       [[(: Passed (False)
Test 985:       [(}{{: Passed (False)
Test 986:       ((((({{}})))): Passed (True)
Test 987:       ([: Passed (False)
Test 988:       [: Passed (False)
Test 989:       {{: Passed (False)
Test 990:       {{()}}: Passed (True)
Test 991:       {}]: Passed (False)
Test 992:       ()){: Passed (False)
Test 993:       {{){): Passed (False)
Test 994:       [}){}){): Passed (False)
Test 995:       ([[(: Passed (False)
Test 996:       [)}: Passed (False)
Test 997:       {]}{]{}: Passed (False)
Test 998:       {[]()[]}: Passed (True)
Test 999:       {[}][: Passed (False)
Test 1000:      (}}}: Passed (False)
Passed 1000 out of 1000 tests
```

2.6.1 Invalid Inputs

Invalid inputs are a superset of valid inputs, thus the selection of brackets to insert at any given point is expanded to include all invalid closing brackets as well. The generator then selects a random index i to insert random brackets until the max length is reached. Then, the generator checks if the number of opening brackets match the number of closing brackets for each type of bracket. If they match, a random index is selected again to either insert or remove a bracket. This ensures that we also deal with the case that the length of the input string is odd. The generator then provides the input string and whether the string is valid to a checker function, which compares the output of the developed algorithm with the validity of the input string. To run the tests, a user may run `python test.py --tests \{number of tests\} --length \{max length of input strings\}`. The output will show how many tests the algorithm passes, and what the generated input strings were for each test.

2.7 Listings

2.7.1 Algorithm

```python
import re

def main(statement:str):
  return is_balanced(statement)

def is_balanced(statement:str) -> bool:
  bracket_pairing = {
    "{": "}",
    "[": "]",
    "(": ")"
  }
  # fast check
  statement = re.sub(r"[A-Za-z0-9\*\-\+\^\/\=]", "", statement)
  if len(statement) % 2 != 0: return False
  brackets = [bracket for bracket in bracket_pairing.keys()]\
    + [bracket for bracket in bracket_pairing.values()]
  stack = [ ]
  for char in statement:
    if char in brackets:
      if char in bracket_pairing.keys():
        stack.append(char)
      else:
        try:
          if bracket_pairing[stack.pop()] != char:
            return False
        except IndexError:
          return False
  return len(stack) == 0

if __name__ == "__main__":
  res = main(input("Enter an algebraic statement: "))
  print(res)
```

2.7.2 Testing

```python
import argparse
import random
from main import is_balanced

def test_is_balanced(iters: int, max_length:int=10):
  """Tests the is_balanced function over a given number of iterations.

  Args:
    iters (int): number of iterations
  """
  results = []
  for i in range(iters):
    statement, proper = statement_generator(random.randint(1, max_length))
    print(f"Test {i+1}:\t{statement}:", end=" ")
    res = is_balanced(statement)
    if res == proper:
      print(f"Passed ({res})")
    else:
      print(f"Failed: {res} (should be {proper})")
    results.append(res == proper)
  print(f"Passed {results.count(True)} out of {iters} tests")

def statement_generator(length: int):
  """Generates a random algebraic statement of a given length.

  Args:
    length (int): length of the statement

```

```
    Returns:
      str: random algebraic statement
    """
    length //= 2
    bracket_pairing = {
      "{": "}",
      "[": "]",
      "(": ")"
    }
    brackets = [bracket for bracket in bracket_pairing.keys()]\
      + [bracket for bracket in bracket_pairing.values()]
    ret = ""
    state = random.choice([True, False])
    ret += random.choice([bracket for bracket in bracket_pairing.keys()])
    stack = [ret[0]]
    for _ in range(length):
      if state:
        candidates = [b for b in bracket_pairing.keys()]
        if ret[-1] in bracket_pairing.keys():
          candidates += [bracket_pairing[ret[-1]]]
        ret += random.choice(candidates)
        if ret[-1] in bracket_pairing.values():
          stack.pop()
        else:
          stack.append(ret[-1])
      else:
        ret += random.choice(brackets)
    for _ in range(len(stack)):
      if state:
        ret += bracket_pairing[stack.pop()]
      else:
        ret += random.choice(brackets)
    if not state:
      n_additions = random.randint(0, length)
      insertion_index = random.randint(0, len(ret))
      additions = [random.choice(brackets) for _ in range(n_additions)]
      ret = ret[:insertion_index-1] + "".join(additions) + ret[insertion_index+1:len(ret)+1-n_additions]
      # count the number of bracket pairs
      bracket_counts = {(k, v): 0 for k, v in bracket_pairing.items()}
      for char in ret:
        for k, v in bracket_pairing.items():
          if char == k:
            bracket_counts[(k, v)] += 1
          elif char == v:
            bracket_counts[(k, v)] -= 1
      # if the statement is potentially balanced, either remove or add a random character.
      if all([count == 0 for count in bracket_counts.values()]):
        # remove a random character
        if len(ret) > 2:
          loc = random.randint(1, len(ret)-1)
          ret = ret[:loc] + ret[loc+1:]
        else:
          ret += random.choice(brackets)
    return (ret, state)

  def main(tests: int=1000, max_length:int=10):
    test_is_balanced(tests, max_length)

  if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("-t", "--tests", type=int, default=1000, help="number of tests to run")
    parser.add_argument("-l", "--length", type=int, default=10, help="maximum length of the statement")
    args = parser.parse_args()
    main(args.tests, args.length)
```

Question 3

3.1 Problem Statement

3.2 Requirements/Specification

3.3 User Guide

3.4 Structure/Design

3.5 Limitations

3.6 Testing

3.7 Listings

Question 4

4.1 Problem Statement

Write a recursive algorithm to check that a sentence is a palindrome (ignoring blanks, lower case and upper case differences, and punctuation marks, so that “Madam, I’m Adam” is accepted as a palindrome). Analyse the efficiency of your implementation and provided a detailed discussion of its time complexity.

Example 4.1.1

Please enter a sentence: Madam, I’m Adam Check if “Madam, I’m Adam” is a palindrome: True

4.2 Requirements/Specification

This program is supposed to compare the characters in the sentence. First and last, Second and second last etc. If it matches, it is a palindrome. Some assumptions/conditions would be ignoring blanks, lower case, upper case differences, and punctuation marks. Empty strings will be considered as a palindrome too.

4.3 User Guide

1. Click on the “Run” button in the IDE to run the program with python. Alternatively, running `python file.py` will run the program.
2. Input a sentence when prompted in the command line interface.
3. The resulting output will show whether the input sentence was a palindrome.

4.4 Structure/Design

The design of the system is such that after it recieves an input from the user, it will remove all punctuation and whitespaces in the string, then change all uppercase characters to lowercase characters. The algorithm will then check the first and last characters in the string to see if they match. If they do, the function is recursively called on the manipulated string without the first and last characters. The time complexity of this algorithm is $O(n)$.

4.5 Limitations

The algorithm expects only the ASCII character set as input, and may not work on sentences including UTF-8 characters beyond the ASCII character set.

4.6 Testing

4.7 Listings

Question 5

5.1 Problem Statement

5.2 Requirements/Specification

5.3 User Guide

5.4 Structure/Design

5.5 Limitations

5.6 Testing

5.7 Listings