CSC 421: Algorithm Design and Analysis

Spring 2019

Brute force approach & efficiency

- brute force design
- KISS vs. generality
- exhaustive search: string matching
- generate & test: N-queens, TSP, Knapsack
- inheritance & efficiency
  - ArrayList → SortedArrayList

HW from 321

HW1: Credit Card Verification

According to the American Banking Association, there were 364 million open credit card accounts in the U.S. by the end of 2017. While credit cards have become the preferred method of payment for American consumers, few ever wonder about the random-seeming sequence of digits on the front. In fact, these digits are far from random. Visa, Mastercard and Discover all utilize a 16-digit sequence, with the first six digits identifying the card issuer and the next nine digits identifying the user's account number. American Express utilizes a 15-digit sequence, with six digits for the issuer and eight for the user account number. The remaining digit on each card serves as a check number for verifying the card's validity. This last digit is assigned according to the Luhn Formula, which is nicely described in this page from creditcards.com.

PART 1: Verifying Sequences (2-person team)

For the first part of this assignment, you are to write a Java program that reads in a series of digit sequences from a file whose name is entered by the user. Your program must identify whether each is a valid credit card sequence, i.e., whether it is of the correct format according to the Luhn Formula. Your program should display each sequence on a line, followed by its classification: either VALID or INVALID. Note: there will be one digit sequence per line, and there may be spaces within the sequence (which are ignored with respect to the Luhn Formula). Any sequence that contains a character other than a digit or space is considered invalid.

For example, suppose the file specified by the user contained the following:

```
4289 0298 7524 0023
4289 0298 7524 0026
313 4890 444 2000 120
42 89 01 44 32 58 99 4
42 89 01 44 32 58 99 40
4289 0144 3258 9941
1234-5678-9876-5432
```

Then, your program should output:

```
4289 0298 7524 0023 VALID
4289 0298 7524 0026 INVALID
313 4890 444 2000 120 VALID
42 89 01 44 32 58 99 4 VALID
42 89 01 44 32 58 99 40 INVALID
4289 0144 3258 9941 INVALID
1234-5678-9876-5432 INVALID
```

For Part 1, you will work in a two-person team with your partner assigned by the instructor. The purpose of this assignment is to refamiliarize you with Java programming and identify any holes in your knowledge/skills. You may not consult with classmates or persons outside your team (other than the instructor, of course). You must demonstrate good programming style when completing this assignment, including the appropriate use of Javadoc-style comments for all classes and methods. The interface for your program is entirely up to you. It need not be fancy, but it must *at least* allow the user to enter a file name and view the results. You may choose to use the GUI builder that comes with NetBeans if you wish.

```
public class CardDriver {
    public static void main(String[] args) {
        System.out.print("Enter the file of credit card numbers: ");
        Scanner input = new Scanner(System.in);
        String filename = input.next();
        System.out.println();

        try {
            Scanner infile = new Scanner(new File(filename));
            while (infile.hasNextLine()) {
                String number = infile.nextLine().trim();
                String cleanNum = number.replace(" ", "");

                boolean validFormat = true;
                for (int i = 0; i < cleanNum.length(); i++) {
                    if (!Character.isDigit(cleanNum.charAt(i))) {
                        validFormat = false;
                    }
                }

                int sum = 0;
                for (int i = 0; i < cleanNum.length(); i++) {
                    if (cleanNum.length() % 2 == i % 2) {
                        int twiceValue = 2 * (cleanNum.charAt(i) - '0');
                        if (twiceValue > 9) {
                            twiceValue = 1 + (twiceValue % 10);
                        }
                        sum += twiceValue;
                    } else {
                        sum += (cleanNum.charAt(i) - '0');
                    }
                }
                boolean validSequence = (sum % 10 == 0);

                if (validFormat && validSequence) {
                    System.out.println(number + " VALID");
                } else {
                    System.out.println(number + " INVALID");
                }
            }
        } catch (java.io.FileNotFoundException e) {
            System.out.println("File not found.");
        }
    }
}
```

Brute force design

- prompt for file name
- read in each card number from file
- remove spaces and check for non-digits
- check digits using Luhn Formula
- display status of each number

3

Adding modularity

this brute force approach works, but is difficult to read and modify

- one simple step is to add modularity (via helper methods)

```
    private static boolean isValidFormat(String num) {
        for (int i = 0; i < num.length(); i++) {
            if (!Character.isDigit(num.charAt(i))) {
                return false;
            }
        }
        return true;
    }


    private static boolean isValidSequence(String num) {
        int sum = 0;
        for (int i = 0; i < num.length(); i++) {
            if (num.length() % 2 == i % 2) {
                int twiceValue = 2 * (num.charAt(i) - '0');
                if (twiceValue > 9) {
                    twiceValue = 1 + (twiceValue % 10);
                }
                sum += twiceValue;
            }
            else {
                sum += (num.charAt(i) - '0');
            }
        }

        return sum % 10 == 0;
    }
```

4

Modular version

helper methods make the main method simpler, more readable

```
public class CardDriver {
    public static void main(String[] args) {
        System.out.print("Enter the file of credit card numbers: ");
        Scanner input = new Scanner(System.in);
        String filename = input.next();
        System.out.println();

        try {
            Scanner infile = new Scanner(new File(filename));
            while (infile.hasNextLine()) {
                String number = infile.nextLine().trim();
                String cleanNum = number.replace(" ", "");

                if (CardDriver.isValidFormat(cleanNum) &&
                        CardDriver.isValidSequence(cleanNum)) {
                    System.out.println(number + " VALID");
                }
                else {
                    System.out.println(number + " INVALID");
                }
            }
        }
        catch (java.io.FileNotFoundException e) {
            System.out.println("File not found.");
        }
    }

    // HELPER METHODS HERE

}
```

5

Adding objects

object-oriented approach: identify objects that can be modeled with code

- classes encapsulate the data and behaviors of objects
- easier to develop, test, and reuse

```
public class CardNumber {
    private String number;
    private String cleanNum;

    public CardNumber(String num) {
        this.number = num.trim();
        this.cleanNum = this.number.replace(" ", "");
    }

    public boolean isValidFormat() {
        // SIMILAR TO PREVIOUSLY SHOWN METHOD
    }

    public boolean isValidSequence() {
        // SIMILAR TO PREVIOUSLY SHOWN METHOD
    }

    public String toString() {
        return number;
    }
}
```

6

OO version

can then define a separate, simple driver class

```
public class CardDriver {
    public static void main(String[] args) {
        System.out.print("Enter the file of credit card numbers: ");
        Scanner input = new Scanner(System.in);
        String filename = input.next();
        System.out.println();

        try {
            Scanner infile = new Scanner(new File(filename));
            while (infile.hasNextLine()) {
                CardNumber number = new CardNumber(infile.nextLine());
                if (number.isValidFormat() && number.isValidSequence()) {
                    System.out.println(number + " VALID");
                }
                else {
                    System.out.println(number + " INVALID");
                }
            }
        }
        catch (java.io.FileNotFoundException e) {
            System.out.println("File not found.");
        }
    }
}
```

7

Brute force

if you have a one-time task of checking a file of credit card numbers,

- any of these three approaches is fine
- it could be argued that helper methods and/or classes will speed up testing & debugging, so worth it

in the real-world, problem-solving is rarely a one-time task

- you may need to go back and revise your solution as specifications change
- you may want to reuse parts of your solution

PART 2: Organizing Output & Error Detection (independent work)

For the second part of this assignment, you are to organize the output so that all of the valid sequences are displayed together (with heading VALID), followed by all of the invalid sequences (with heading INVALID). Within each category, the sequences should be displayed in increasing numerical order (ignoring any non-digits).

You should also augment your program so that it can handle sequences in which a single digit has been corrupted. That is, the character '?' may appear in a sequence in the place of an unknown digit. Because of the check number, it should be possible to determine the value of the missing digit. Augment your program so that it determines the missing digit and adds the corresponding sequence to the list of valid sequences. Note: any code that contains more than one '?' is considered invalid.

8

Extension

the OO version is easily extensible

to handle ordering, implement the Comparable interface

to handle error correction, add a method for fixing a corrupted digit

```
public class CardNumber implements Comparable<CardNumber> {
    private String number;
    private String cleanNum;

    public CardNumber(String num) {
        this.number = num.trim();
        this.cleanNum = this.number.replace(" ", "");

        if (this.number.indexOf('?') != -1 &&
              this.number.indexOf('?') == this.number.lastIndexOf('?')) {
            this.fixNumber();
        }
    }

    public boolean isValidFormat() {
        // AS BEFORE
    }

    public boolean isValidSequence() {
        // AS BEFORE
    }

    public String toString() {
        return number;
    }

    public int compareTo(CardNumber other) {
        return this.cleanNum.compareTo(other.cleanNum);
    }

    ////////////////////////////////////////////////////////////////////

    private void fixNumber() {
        String safeNum = this.number;

        for (int i = 0; i < 10; i++) {
            this.number = safeNum.replace('?', (char)('0'+i));
            this.cleanNum = this.number.replace(" ", "");
            if (this.isValidFormat() && this.isValidSequence()) {
                return;
            }
        }

        this.number = safeNum;
        this.cleanNum = this.number.replace(" ", "");
    }
}
```

9

Extension

add a CardChecker class that will separate valid & invalid numbers

since CardNumbers are Comparable, can use Collections.sort to order the two lists

note: neither class does I/O

```
public class CardChecker {
    private ArrayList<CardNumber> validNumbers;
    private ArrayList<CardNumber> invalidNumbers;

    public CardChecker() {
        this.validNumbers = new ArrayList<CardNumber>();
        this.invalidNumbers = new ArrayList<CardNumber>();
    }

    public void storeCC(CardNumber number) {
        if (number.isValidFormat() && number.isValidSequence()) {
            this.validNumbers.add(number);
        }
        else {
            this.invalidNumbers.add(number);
        }
    }

    public String toString() {
        return "VALID\n" + this.stringify(this.validNumbers) +
                "\nINVALID\n" + this.stringify(this.invalidNumbers);
    }


////////////////////////////////////////////////////////////////////

    private String stringify(ArrayList<CardNumber> nums) {
        Collections.sort(nums);
        String message = "";
        for (CardNumber num : nums) {
            message += num + "\n";
        }
        return message;
    }
}
```

10

Extension

the driver is solely responsible for I/O

all program logic is handled by objects

Model-View-Controller pattern

- model implements the logic of the solution (here, CardNumber & CardChecker)
- view is the interface the user uses (here, terminal window & keyboard)
- controller connects the two, performs I/O (here, CardDriver)

```
public class CardDriver {
    public static void main(String[] args) {
        System.out.print("Enter the file of credit card numbers: ");
        Scanner input = new Scanner(System.in);
        String filename = input.next();
        System.out.println();

        try {
            CardChecker checker = new CardChecker();

            Scanner infile = new Scanner(new File(filename));
            while (infile.hasNext()) {
                String numStr = infile.nextLine();
                checker.storeCC(new CardNumber(numStr));
            }

            System.out.println(checker);
        }
        catch (java.io.FileNotFoundException e) {
            System.out.println("File not found.");
        }
    }
}
```

11

Brute force

many algorithms can be characterized as *brute force* as well

- utilize a straightforward approach, maybe not the most efficient or extensible
- consider the exponentiation application

  simple, iterative version: $a^b = a * a * a * \ldots * a$ (b times)
  recursive version: $a^b = a^{b/2} * a^{b/2}$

  while the recursive version is more efficient, O(log N) vs. O(N), is it really worth it?

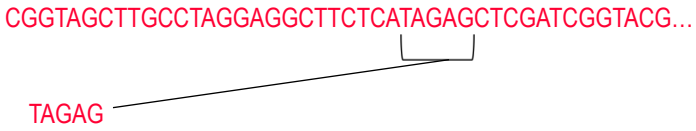
brute force works fine when

- the problem size is small
- only a few instances of the problem need to be solved
- need to build a prototype to study the problem

12

Exhaustive search: string matching

- consider the task of the String indexOf method
  find the first occurrence of a desired substring in a string
- this problem occurs in many application areas, e.g., DNA sequencing



Exhaustive string matching

the brute force/exhaustive approach is to sequentially search

CGGTAGCTTGCCTAGGAGGCTTCTCATAGAGCTCGATCGGTACG...
CGGTAGCTTGCCTAGGAGGCTTCTCATAGAGCTCGATCGGTACG...
CGGTAGCTTGCCTAGGAGGCTTCTCATAGAGCTCGATCGGTACG...
CGGTAGCTTGCCTAGGAGGCTTCTCATAGAGCTCGATCGGTACG...
…
CGGTAGCTTGCCTAGGAGGCTTCTCATAGAGCTCGATCGGTACG...

```
public static int indexOf(String seq, String desired) {
  for (int start = 0; start <= seq.length() - desired.length(); start++) {
    String sub = seq.substring(start, start+desired.length());
    if (sub.equals(desired)) {
      return start;
    }
  }
  return -1;
}
```

efficiency of search? we can do better (more later) – do we need to?

Generate & test

sometimes exhaustive algorithms are referred to as "generate & test"

- can express algorithm as generating each candidate solution systematically, testing each to see if the candidate is actually a solution

string matching: try seq.substring(0, desired.length())
if no match, try seq.substring(1, desired.length()+1)
if no match, try seq.substring(2, desired.length()+2)
…

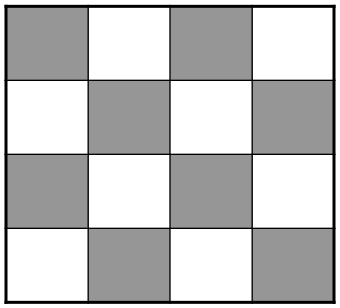
extreme (and extremely bad) example – permu-sort

- to sort a list of items, generate every permutation and test to see if in order
- efficiency?

15

Generate & test: N-queens

given an NxN chess board, place a queen on each row so that no queen is in jeopardy



generate & test approach

- systematically generate every possible arrangement
- test each one to see if it is a valid solution

this will work (in theory), but the size of the search space may be prohibitive

4x4 board → $\binom{16}{4}$ = 1,820 arrangements
4! = 24 arrangements

8x8 board → $\binom{64}{8}$ = 131,198,072 arrangements
8! = 40,320 arrangements

again, we can do better (more later)

16

nP-hard problems: traveling salesman

there are some problems for which there is no known "efficient" algorithm (i.e., nothing polynomial) → known as nP-hard problems (more later)

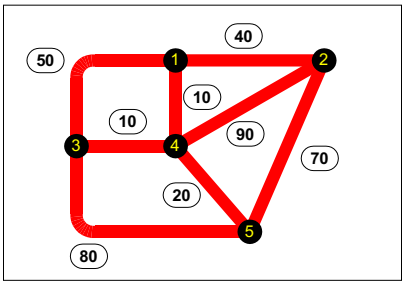
generate & test may be the only option

Traveling Salesman Problem: A salesman must make a complete tour of a given set of cities (no city visited twice except start/end city) such that the total distance traveled is minimized.

example: find the shortest tour given this map

generate & test → try every possible route

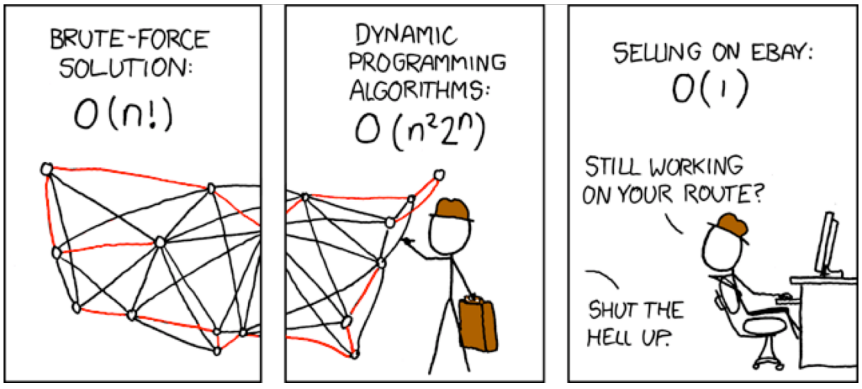
efficiency?



17

xkcd: Traveling Salesman Problem comic

a dynamic programming approach (more later) can improve performance slightly, but still intractable for reasonably large N



18

nP-hard problems: knapsack problem

another nP-hard problem:

Knapsack Problem: Given N items of known weights $w_1,\ldots,w_N$ and values $v_1,\ldots,v_N$ and a knapsack of capacity W, find the highest-value subset of items that fit in the knapsack.

example: suppose a knapsack with capacity of 50 lb. Which items do you take?

tiara	$5000	3 lbs
coin collection	$2200	5 lbs
HDTV	$2100	40 lbs
laptop	$2000	8 lbs
silverware	$1200	10 lbs
stereo	$800	25 lbs
PDA	$600	1 lb
clock	$300	4 lbs

generate & test solution:

- try every subset & select the one with greatest value

19

Dictionary revisited

recall the Dictionary class earlier

- the ArrayList add method simply appends the item at the end → O(1)
- the ArrayList contains method performs sequential search → O(N)

this is OK if we are doing lots of adds and few searches

```
import java.util.List;
import java.util.ArrayList;
import java.util.Scanner;
import java.io.File;

public class Dictionary {
    private List<String> words;

    public Dictionary() {
        this.words = new ArrayList<String>();
    }

    public Dictionary(String filename) {
        this();

        try {
            Scanner infile = new Scanner(new File(filename));
            while (infile.hasNext()) {
                String nextWord = infile.next();
                this.words.add(nextWord.toLowerCase());
            }
        }
        catch (java.io.FileNotFoundException e) {
            System.out.println("FILE NOT FOUND");
        }
    }

    public void add(String newWord) {
        this.words.add(newWord.toLowerCase());
    }

    public void remove(String oldWord) {
        this.words.remove(oldWord.toLowerCase());
    }

    public boolean contains(String testWord) {
        return this.words.contains(testWord.toLowerCase());
    }
}
```

20

StopWatch

big-Oh analysis is good for understanding long-term growth

sometimes, you want absolute timings to compare algorithm performance on real data

```
public class StopWatch {
    private long lastStart;
    private long lastElapsed;
    private long totalElapsed;

    public StopWatch() {
        this.reset();
    }

    public void start() {
        this.lastStart = System.currentTimeMillis();
    }

    public void stop() {
        long stopTime = System.currentTimeMillis();
        if (this.lastStart != -1) {
            this.lastElapsed = stopTime - this.lastStart;
            this.totalElapsed += this.lastElapsed;
            this.lastStart = -1;
        }
    }

    public long getElapsedTime() {
        return this.lastElapsed;
    }

    public long getTotalElapsedTime() {
        return this.totalElapsed;
    }

    public void reset() {
        this.lastStart = -1;
        this.lastElapsed = 0;
        this.totalElapsed = 0;
    }
}
```

21

Timing dictionary searches

we can use our `StopWatch` class to verify the O(N) efficiency

dict. size	build time
38,621	401 msec
77,242	612 msec
144,484	1123 msec

dict. size	search time
38,621	1.10 msec
77,242	2.61 msec
144,484	5.01 msec

execution time roughly doubles as dictionary size doubles

```
import java.util.Scanner;
import java.io.File;

public class DictionaryTimer {

  public static void main(String[] args) {
      System.out.println("Enter name of dictionary file:");
      Scanner input = new Scanner(System.in);
      String dictFile = input.next();

      StopWatch timer = new StopWatch();

      timer.start();
      Dictionary dict = new Dictionary(dictFile);
      timer.stop();

      System.out.println(timer.getElapsedTime());

      timer.start();
      for (int i = 0; i < 100; i++) {
          dict.contains("zzyzyba");
      }
      timer.stop();

      System.out.println(timer.getElapsedTime()/100.0);
  }
}
```

22

Sorting the list

if searches were common, then we might want to make use of binary search

- this requires sorting the words first, however

we could change the Dictionary class to do the sorting and searching

- a more general solution would be to extend the ArrayList class to SortedArrayList
- could then be used in any application that called for a sorted list

recall:

```
public class java.util.ArrayList<E> implements List<E> {
    public ArrayList() { … }
    public boolean add(E item) { … }
    public void add(int index, E item) { … }
    public E get(int index) { … }
    public E set(int index, E item) { … }
    public int indexOf(Object item) { … }
    public boolean contains(Object item) { … }
    public boolean remove(Object item) { … }
    public E remove(int index) { … }
    …
}
```

23

SortedArrayList (v.1)

using inheritance, we only need to redefine what is new

- `add` method sorts after adding; `indexOf` uses binary search
- no additional fields required

- big-Oh for `add`? big-Oh for `indexOf`?

```
import java.util.ArrayList;
import java.util.Collections;

public class SortedArrayList<E extends Comparable<? super E>> extends ArrayList<E> {
    public SortedArrayList() {
        super();
    }

    public boolean add(E item) {
        super.add(item);
        Collections.sort(this);
        return true;
    }

    public int indexOf(Object item) {
        return Collections.binarySearch(this, (E)item);
    }
}
```

24

SortedArrayList (v.2)

is this version any better? when?

- big-Oh for `add`?
- big-Oh for `indexOf`?

```
import java.util.ArrayList;
import java.util.Collections;

public class SortedArrayList<E extends Comparable<? super E>> extends ArrayList<E> {
    public SortedArrayList() {
        super();
    }

    public boolean add(E item) {        // NOTE: COULD REMOVE THIS METHOD AND
        super.add(item);                // JUST INHERIT THE ADD METHOD FROM
        return true;                    // ARRAYLIST AS IS
    }

    public int indexOf(Object item) {
        Collections.sort(this);
        return Collections.binarySearch(this, (E)item);
    }
}
```

25

SortedArrayList (v.3)

if insertions and searches are mixed, sorting for each insertion/search is extremely inefficient

- instead, could take the time to insert each item into its correct position
- big-Oh for `add`? big-Oh for `indexOf`?

```
import java.util.ArrayList;
import java.util.Collections;

public class SortedArrayList<E extends Comparable<? super E>> extends ArrayList<E> {
    public SortedArrayList() {
        super();
    }

    public boolean add(E item) {
        int i;
        for (i = 0; i < this.size(); i++) {
            if (item.compareTo(this.get(i)) < 0) {
                break;
            }
        }
        super.add(i, item);
        return true;
    }

    public int indexOf(Object item) {
        return Collections.binarySearch(this, (E)item);
    }
}
```

search from the start vs. from the end?

26

Dictionary using SortedArrayList

note that repeated calls to add serve as insertion sort

dict. size	build time
38,621	29.2 sec
77,242	127.9 sec
144,484	526.2 sec

dict. size	search time
38,621	0.0 msec
77,242	0.0 msec
144,484	0.1 msec

build time roughly quadruples as dictionary size doubles; search time is trivial

```
import java.util.Scanner;
import java.io.File;
import java.util.Date;

public class DictionaryTimer {

  public static void main(String[] args) {
      System.out.println("Enter name of dictionary file:");
      Scanner input = new Scanner(System.in);
      String dictFile = input.next();

      StopWatch timer = new StopWatch();

      timer.start();
      Dictionary dict = new Dictionary(dictFile);
      timer.stop();

      System.out.println(timer.getElapsedTime());

      timer.start();
      for (int i = 0; i < 100; i++) {
          dict.contains("zzyzyba");
      }
      timer.stop();

      System.out.println(timer.getElapsedTime()/100.0);
  }
}
```

27

SortedArrayList (v.4)

if adds tend to be done in groups (as in loading the dictionary)

- it might pay to perform lazy insertions & keep track of whether sorted
- big-Oh for add? big-Oh for indexOf?
- if desired, could still provide addInOrder method (as before)

```
import java.util.ArrayList;
import java.util.Collections;

public class SortedArrayList<E extends Comparable<? super E>> extends ArrayList<E> {
    private boolean isSorted;

    public SortedArrayList() {
        super();
        this.isSorted = true;
    }

    public boolean add(E item) {
        this.isSorted = false;
        return super.add(item);
    }

    public int indexOf(Object item) {
        if (!this.isSorted) {
            Collections.sort(this);
            this.isSorted = true;
        }
        return Collections.binarySearch(this, (E)item);
    }
}
```

28

Timing the lazy dictionary on searches

modify the Dictionary class to use the lazy SortedArrayList

dict. size	build time
38,621	340 msec
77,242	661 msec
144,484	1113 msec

dict. size	1st search
38,621	10 msec
77,242	61 msec
144,484	140 msec

dict. size	search time
38,621	0.0 msec
77,242	0.0 msec
144,484	0.1 msec

```
import java.util.Scanner;
import java.io.File;
import java.util.Date;

public class DictionaryTimer {
  public static void main(String[] args) {
      System.out.println("Enter name of dictionary file:");
      Scanner input = new Scanner(System.in);
      String dictFile = input.next();

      StopWatch timer = new StopWatch()

      timer.start();
      Dictionary dict = new Dictionary(dictFile);
      timer.stop();
      System.out.println(timer.getElapsedTime());

      timer.start();
      dict.contains("zzyzyba");
      timer.stop();
      System.out.println(timer.getElapsedTime());

      timer.start();
      for (int i = 0; i < 100; i++) {
          dict.contains("zzyzyba");
      }
      timer.stop();
      System.out.println(timer.getElapsedTime()/100.0);
  }
}
```

29