CSC 421: Algorithm Design and Analysis

Spring 2014

Brute force approach & efficiency

- league standings example
- KISS vs. generality
- exhaustive search: string matching
- generate & test: N-queens, TSP, Knapsack
- inheritance & efficiency
  - ArrayList → SortedArrayList

Past HW assignment

Description:

You have been hired to computerize Creighton's intramural basketball records. Each team in the intramural league has a unique name and plays some number of games against other teams in the league (possibly playing the same team more than once). You are to take the scores of all the games played and create the standings for the league, displaying the win-loss records for the teams and ordering them based on winning percentage.

Input:

The input will come from a file, whose name is entered by the user. The file will consist of some number of lines of text of the form

```
NameH ScoreH NameV ScoreV
```

where NameH is the name of the home team, ScoreH is the home team's score, NameV is the name of the visiting team, and ScoreV is the visiting team's score. You may assume that there are no ties, so one of the scores will be larger and signify the winner of that game.

Example Input:

```
Jays 85 Dogs 80
Cats 65 Dogs 70
Squirrels 77 Jays 78
Cats 80 Squirrels 55
```

Output:

For each team listed in the scores, you should output a line containing the team name followed by their win-loss record. There should be one space between the team name and the number of wins, and a single hyphen between the number of wins and losses. Teams should be listed in order based on winning percentage. That is, the team with highest winning percentage should be listed first and the team with the lowest winning percentage last. Among teams with the same number winning percentage, they should be listed in alphabetical order by team name.

Example Output:

```
Jays 2-0
Cats 1-1
Dogs 1-1
Squirrels 0-2
```

Top-level design

```
public class LeagueDriver {
    public static void main(String[] args) {
        System.out.println("Enter a file name: ");
        Scanner input = new Scanner(System.in);

        ScoresList scores = new ScoresList();

        String filename = input.next();
        try {
           Scanner infile = new Scanner(new File(filename));
           while (infile.hasNextLine()) {
               String team1 = infile.next();
               int score1 = infile.nextInt();
               String team2 = infile.next();
               int score2 = infile.nextInt();

               scores.recordScore(team1, score1, team2, score2);
           }

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

at the top-level, need to

- read in games scores
- store them in some structure
- then display the record for each team

League standings (v. 1)

```
public class ScoresList {
    private Map<String, WinLossRecord> scores;

    public ScoresList() {
        this.scores = new HashMap<String, WinLossRecord>();
    }

    public void recordScore(String team1, int score1,
                            String team2, int score2) {
        if (!this.scores.containsKey(team1)) {
            this.scores.put(team1, new WinLossRecord());
        }
        if (!this.scores.containsKey(team2)) {
            this.scores.put(team2, new WinLossRecord());
        }

        if (score1 > score2) {
            this.scores.get(team1).addWin();
            this.scores.get(team2).addLoss();
        }
        else {
            this.scores.get(team2).addWin();
            this.scores.get(team1).addLoss();
        }

    }

    public void displayScores() {
        for (String team : this.scores.keySet()) {
            System.out.println(team + ": " + this.scores.get(team));
        }
    }
}
```

a Map is a natural structure for keeping team-record pairs

if we don't care about order, HashMap is fine (and fast!)

if we use a TreeMap, then teams will be displayed in alphabetical order

League standings (v. 1)

```
public class WinLossRecord {
    private int numWins;
    private int numLosses;

    public WinLossRecord() {
        this.numWins = 0;
        this.numLosses = 0;
    }

    public void addWin() {
        this.numWins++;
    }

    public void addLoss() {
        this.numLosses++;
    }

    public String toString() {
        return this.numWins + "-" + this.numLosses;
    }
}
```

`WinLossRecord` class provides methods for adding wins & losses

the `toString` method makes it easy to display a `WinLossRecord`

5

How efficient is this solution?

let G = the number of games and T = the number of teams

- while loop executes G times, each time through must
  - ✓read in teams & scores
  - ✓check to see if teams already stored in Map
  - ✓add record to Map if not already stored
  - ✓determine which team won/lost
  - ✓update records of both teams in Map
- displaying the team names & records

6

Top-level design (v. 2)

```
public class LeagueDriver {
    public static void main(String[] args) {
        System.out.println("Enter a file name: ");
        Scanner input = new Scanner(System.in);

        ScoresList scores = new ScoresList();

        String filename = input.next();
        try {
           Scanner infile = new Scanner(new File(filename));
           while (infile.hasNextLine()) {
               String team1 = infile.next();
               int score1 = infile.nextInt();
               String team2 = infile.next();
               int score2 = infile.nextInt();

               scores.recordScore(team1, score1, team2, score2);
           }

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

suppose we want the standings ordered by winning percentage

what, if anything, changes at the top-level?

7

League standings (v. 2)

```
public class ScoresList {
    private Map<String, WinLossRecord> scores;

    public ScoresList() {
        this.scores = new TreeMap<String, WinLossRecord>();
    }

    public void recordScore(String team1, int score1,
                            String team2, int score2) {
        if (!this.scores.containsKey(team1)) {
            this.scores.put(team1, new WinLossRecord(team1));
        }
        if (!this.scores.containsKey(team2)) {
            this.scores.put(team2, new WinLossRecord(team2));
        }
        if (score1 > score2) {
            this.scores.get(team1).addWin();
            this.scores.get(team2).addLoss();
        }
        else {
            this.scores.get(team2).addWin();
            this.scores.get(team1).addLoss();
        }
    }

    public void displayScores() {
        TreeSet<WinLossRecord> recs = new TreeSet<WinLossRecord>();
        for (String team : this.scores.keySet()) {
            recs.add(this.scores.get(team));
        }

        for (WinLossRecord nextRec : recs) {
            System.out.println(nextRec);
        }
    }
}
```

to display the teams by winning percentage, generalize WinLossRecord

- to store team name
- to be Comparable

could put into an ArrayList, sort it, then traverse & display

could put into a TreeSet, then traverse & display

8

League standings (v. 2)

```
public class WinLossRecord implements Comparable<WinLossRecord> {
    private String team;
    private int numWins;
    private int numLosses;

    public WinLossRecord(String team) {
        this.team = team;
        this.numWins = 0;
        this.numLosses = 0;
    }

    public void addWin() {
        this.numWins++;
    }

    public void addLoss() {
        this.numLosses++;
    }

    public String toString() {
        return this.team + ": " + this.numWins + "-" + this.numLosses;
    }

    public int compareTo(WinLossRecord other) {
        double thisPercent = (double)this.numWins/(this.numWins+this.numLosses);
        double otherPercent = (double)other.numWins/(other.numWins+other.numLosses);
        if (thisPercent > otherPercent) {
            return -1;
        }
        else if (thisPercent < otherPercent) {
            return 1;
        }
        else {
            return 0;
        }
    }
```

`WinLossRecord` class provides methods for adding wins & losses

the `toString` method makes it easy to display a `WinLossRecord`

9

How efficient is v. 2?

let G = the number of games and T = the number of teams

- while loop executes G times, each time through must
  - ✓read in teams & scores
  - ✓check to see if teams already stored in Map
  - ✓add record to Map if not already stored
  - ✓determine which team won/lost
  - ✓update records of both teams in Map
- displaying the team names & records
  - ✓get the set of keys
  - ✓get each record & store in TreeSet
  - ✓traverse TreeSet & display

10

Top-level design (v. 3)

```
public class LeagueDriver {
    public static void main(String[] args) {
        System.out.println("Enter a file name: ");
        Scanner input = new Scanner(System.in);

        ScoresList scores = new ScoresList();

        String filename = input.next();
        try {
           Scanner infile = new Scanner(new File(filename));
           while (infile.hasNextLine()) {
               String team1 = infile.next();
               int score1 = infile.nextInt();
               String team2 = infile.next();
               int score2 = infile.nextInt();

               scores.recordScore(team1, score1, team2, score2);
           }

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

suppose we want a tie-breaker for teams with same record

- given same record, list alphabetically by team (as in HW2)

what, if anything, changes at the top-level?

League standings (v. 3)

```
public class ScoresList {
    private Map<String, WinLossRecord> scores;

    public ScoresList() {
        this.scores = new TreeMap<String, WinLossRecord>();
    }

    public void recordScore(String team1, int score1,
                            String team2, int score2) {
        if (!this.scores.containsKey(team1)) {
            this.scores.put(team1, new WinLossRecord(team1));
        }
        if (!this.scores.containsKey(team2)) {
            this.scores.put(team2, new WinLossRecord(team2));

        if (score1 > score2) {
            this.scores.get(team1).addWin();
            this.scores.get(team2).addLoss();
        }
        else {
            this.scores.get(team2).addWin();
            this.scores.get(team1).addLoss();
        }
    }

    public void displayScores() {
        TreeSet<WinLossRecord> recs = new TreeSet<WinLossRecord>();
        for (String team : this.scores.keySet()) {
            recs.add(this.scores.get(team));
        }

        for (WinLossRecord nextRec : recs) {
            System.out.println(nextRec);
        }
    }
}
```

what, if anything, changes in the ScoresList class?

League standings (v. 3)

```
public class WinLossRecord implements Comparable<WinLossRecord> {
    private String team;
    private int numWins;
    private int numLosses;

    public WinLossRecord(String team) {
        this.team = team;
        this.numWins = 0;
        this.numLosses = 0;
    }

    public void addWin() {
        this.numWins++;
    }

    public void addLoss() {
        this.numLosses++;
    }

    public String toString() {
        return this.team + ": " + this.numWins + "-" + this.numLosses;
    }

    public int compareTo(WinLossRecord other) {
        double thisPercent = (double)this.numWins/(this.numWins+this.numLosses);
        double otherPercent = (double)other.numWins/(other.numWins+other.numLosses);
        if (thisPercent > otherPercent) {
            return -1;
        }
        else if (thisPercent < otherPercent) {
            return 1;
        }
        else {
            return this.team.compareTo(other.team);
        }
    }
```

a tie-breaker requires only a tiny modification to WinLossRecord

if winning % is same, then compare team names

Brute force

many problems do not require complex, clever algorithms

- a *brute force* (i.e., straightforward) approach may suffice
- consider the exponentiation application

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

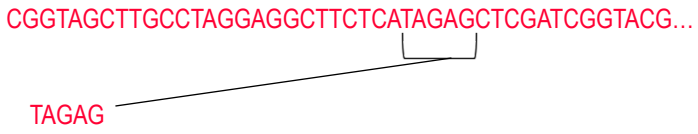
  while the recursive version is more efficient, O(log N) vs. O(N), is it really worth it?

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

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



15

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?

16

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 example – for league standings problem, suppose you knew there was a small, fixed number of games (e.g., 20)

- instead of sorting the records by winning percentage, store records in a TreeMap (ordered by team name)

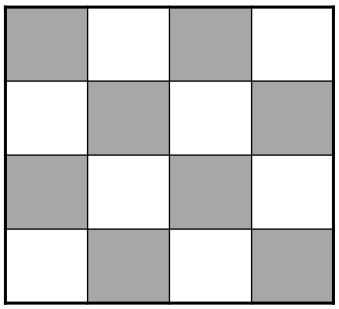
DON'T DO THIS!

traverse the teams, print each 20-0 team
traverse the teams again, print each 19-1 team
…
traverse the teams again, print each 0-20 team

17

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)

18

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)

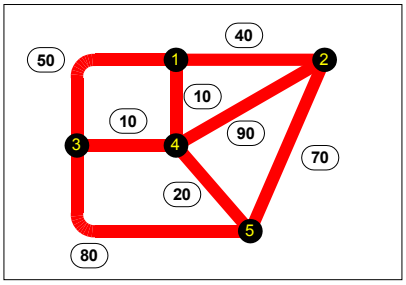
generate & test may be the only option

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

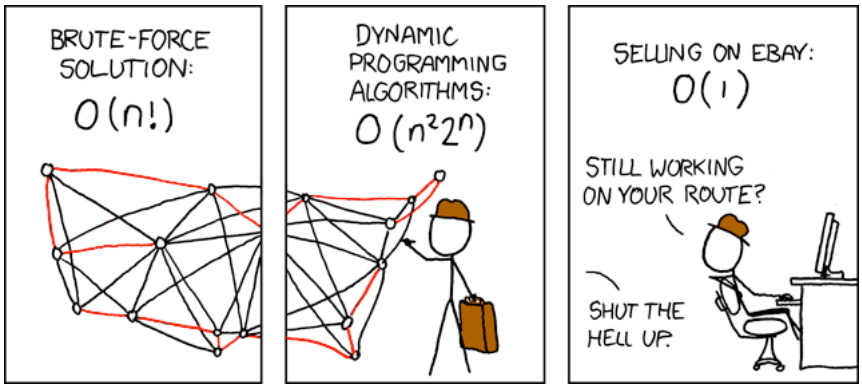
efficiency?



19

xkcd: Traveling Salesman Problem comic

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



20

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

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());
    }
}
```

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;
    }
}
```

23

Timing dictionary searches

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

dict. size	insert 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);
  }
}
```

24

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) { … }
    …
}
```

25

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);
    }
}
```

26

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);
    }
}
```

27

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?

28

Dictionary using SortedArrayList

note that repeated calls to add serve as insertion sort

dict. size	insert 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

insertion 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);
  }
}
```

29

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);
    }
}
```

30

Timing the lazy dictionary on searches

modify the Dictionary class to use the lazy SortedArrayList

dict. size	insert 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);
  }
}
```

31