CSC 421: Algorithm Design and Analysis

Spring 2018

Brute force approach & efficiency

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

HW1 from a previous semester

For this assignment, you will define two related Java programs that process a file of invoice data. An invoice file will consist of invoice records, one invoice per line, which specify the sales date, the customer ID, the number of units sold, the price per unit, and the salesperson ID. Individual entries on a line are separated by whitespace. For example,

```
01-01-2015      Smith1024       100     8.99    TWalker
01-06-2015      Zander99        1500    5.99    SMiller
01-01-2015      Smith1024       200     8.99    JVaska
12-20-2014      Bell222         1       12.99   TWalker
```

You may assume that no two invoices will have the same date, customer ID, and salesperson ID. That is, if a customer makes multiple purchases on the same day from the same salesperson, then those purchases are combined into a single invoice for that day. The customer may have separate invoices on the same day if they are purchased through different salespersons.

Part A: Ordering and Totaling

Write a program named `TotalSales` that reads in invoices from a file (whose name is entered by the user) and displays the invoices, ordered by date, followed by the total number of units sold and the total sales amount. Invoices from the same day may be listed in any order. For example, your program interaction might appear as below (with user input in bold):

```
Enter the invoice file name: invoices.txt

12-20-2014 Bell222 1 12.99 TWalker
01-01-2015 Zander99 1500 5.99 SMiller
01-01-2015 Smith1024 200 8.99 JVaska
01-10-2015 Smith1024 100 8.99 TWalker
Total units = 1801
Total sales = $11694.99
```

In completing this task, you may choose to make use of the provided Invoice class, which stores and accesses the entries in an invoice. You may choose to modify this file as desired, e.g., extending it to implement the `Comparable` interface.

```
public class TotalSales1 {
  public static void main(String[] args) {
    ArrayList<String> invoices = new ArrayList<String>();

    System.out.print("Enter the sales file name: ");
    Scanner input = new Scanner(System.in);
    String filename = input.next();
    try {
      int totalUnits = 0;
      double totalSales = 0.0;
      Scanner infile = new Scanner(new File(filename));
      while (infile.hasNext()) {
        String date = infile.next();
        String customer = infile.next();
        int units = infile.nextInt();
        double price = infile.nextDouble();
        String salesPerson = infile.next();

        String newInvoice = date+" "+customer+" "+units+" "+price+" "+salesPerson;
        TotalSales1.addInOrder(invoices, newInvoice);
        totalUnits += units;
        totalSales += units * price;
      }
      infile.close();

      System.out.println();
      for (String nextInvoice : invoices) {
        System.out.println(nextInvoice);
      }
      System.out.println("Total units = " + totalUnits);
      System.out.println("Total sales = $" + totalSales);
    }
    catch (java.io.FileNotFoundException e) {
      System.out.println("Sales file not found.");
    }
    input.close();
  }
```

Top-level design

- read in invoice data from file
- store them
- calculate total units & sales
- display the invoices, SORTED BY DATE, and the totals

3

(Ugly) helper methods

this brute force approach requires adding the invoices in order (or sorting)

- which requires writing a method to compare dates (or using a Java library class)

```
  private static void addInOrder(ArrayList<String> invoices, String newInvoice) {
    invoices.add(newInvoice);
    int index = invoices.size()-1;
    while (index > 0 && TotalSales1.comesBefore(invoices.get(index).split(" ")[0],
                                                invoices.get(index-1).split(" ")[0])) {
      invoices.set(index, invoices.get(index-1));
      invoices.set(index-1, newInvoice);
      index--;
    }
  }

  private static boolean comesBefore(String date1, String date2) {
    String dateParts1[] = date1.split("-");
    String dateParts2[] = date2.split("-");

    if (Integer.parseInt(dateParts1[2]) != Integer.parseInt(dateParts2[2])) {
      return Integer.parseInt(dateParts1[2]) < Integer.parseInt(dateParts2[2]);
    }
    else if (Integer.parseInt(dateParts1[0]) != Integer.parseInt(dateParts2[0])) {
      return Integer.parseInt(dateParts1[0]) < Integer.parseInt(dateParts2[0]);
    }
    else {
      return Integer.parseInt(dateParts1[1]) < Integer.parseInt(dateParts2[1]);
    }
  }
}
```

4

More OO version

OO philosophy: identify objects that can be modeled with code

- implement and test them separately, can then be reused

`Invoice` class was provided – modify it to implement `Comparable`

```
public class Invoice implements Comparable<Invoice> {

  // FIELDS, CONSTRUCTORS & METHODS AS PROVIDED IN ASSIGNMENT

  public int compareTo(Invoice other) {
    String[] thisParts = this.getDate().split("-");
    String[] otherParts = other.getDate().split("-");
    if (!thisParts[2].equals(otherParts[2])) {
      return Integer.parseInt(thisParts[2]) - Integer.parseInt(otherParts[2]);
    }
    else if (!thisParts[0].equals(otherParts[0])) {
      return Integer.parseInt(thisParts[0]) - Integer.parseInt(otherParts[0]);
    }
    else if (!thisParts[1].equals(otherParts[1])) {
      return Integer.parseInt(thisParts[1]) - Integer.parseInt(otherParts[1]);
    }
    else {
      String thisExtra = this.getCustomerID()+" "+this.getSalesPerson();
      String otherExtra = other.getCustomerID()+" "+other.getSalesPerson();
      return thisExtra.compareTo(otherExtra);
    }
  }
}
```

WHY THE ELSE CASE?

5

More OO version (cont.)

```
public class TotalSales2 {
  public static void main(String[] args) {
    TreeSet<Invoice> invoices = new TreeSet<Invoice>();

    System.out.print("Enter the sales file name: ");
    Scanner input = new Scanner(System.in);
    String filename = input.next();
    try {
      int totalUnits = 0;
      double totalSales = 0.0;
      Scanner infile = new Scanner(new File(filename));
      while (infile.hasNext()) {
        Invoice inv = new Invoice(infile);
        invoices.add(inv);
        totalUnits += inv.getUnits();
        totalSales += inv.getUnits()*inv.getPricePerUnit();
      }
      infile.close();

      System.out.println();
      for (Invoice i : invoices) {
        System.out.println(i);
      }
      System.out.println("Total units = " + totalUnits);
      System.out.println("Total sales = $" + totalSales);
    }
    catch (java.io.FileNotFoundException e) {
      System.out.println("Sales file not found.");
    }
    input.close();
  }
}
```

CLEANER:

- I/O and comparison details are hidden

uses a TreeSet to order the Invoices

- explains the else case
- can't have duplicates

could have used a List

- call Collections.sort once all Invoices have been added

6

Even more OO version

would be even better to remove the Set/List details

- create an InvoiceList class that you can add Invoices to and access as needed

```
public class InvoiceList {
  private Set<Invoice> invoices;
  private int totalUnits;
  private double totalSales;

  public InvoiceList() {
    this.invoices = new TreeSet<Invoice>();
    this.totalUnits = 0;
    this.totalSales = 0.0;
  }

  public void add(Invoice inv) {
    this.invoices.add(inv);
    this.totalUnits += inv.getUnits();
    this.totalSales += inv.getUnits() * inv.getPricePerUnit();
  }

  public String toString() {
    String invStr = "";
    for (Invoice inv : invoices) {
      invStr += inv + "\n";
    }
    return invStr + "Total units = " + totalUnits + "\n" +
                    "Total sales = $" + totalSales;
  }
}
```

a truly useful class would have other methods, e.g., getters & setters

7

Even more OO version (cont.)

```
public class TotalSales3 {
  public static void main(String[] args) {
    InvoiceList invoices = new InvoiceList();

    System.out.print("Enter the sales file name: ");
    Scanner input = new Scanner(System.in);
    String filename = input.next();
    try {
      Scanner infile = new Scanner(new File(filename));
      while (infile.hasNext()) {
        Invoice inv = new Invoice(infile);
        invoices.add(inv);
      }
      infile.close();

      System.out.println(invoices);
    }
    catch (java.io.FileNotFoundException e) {
      System.out.println("Sales file not found.");
    }
    input.close();
  }
}
```

CLEANEST:

- data structure choices are hidden – could be changed without affecting main
- all I/O is contained in the driver – could be changed without affecting classes

is it worth the effort?

- v.2 definitely!
- v.3 yes if reuse is possible

8

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?

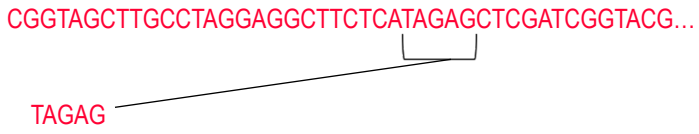
brute force works fine when

- 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

9

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



10

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?

11

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

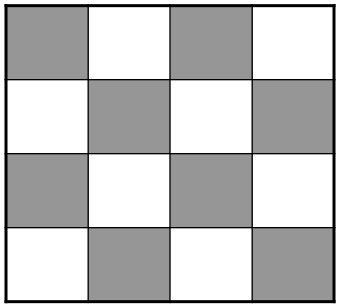
string matching: try seq.substring(0, desired.length())
if no match, try seq.substring(1, desired.length()+1)
if no match, try seq.substring(2, desired.length()+2)
...

extreme (and extremely bad) example – permu-sort

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

12

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)

13

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)

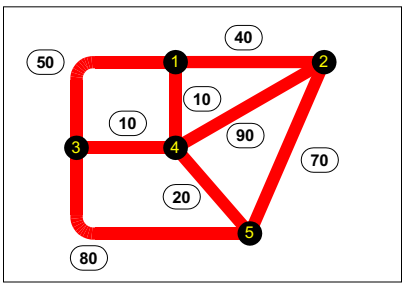
generate & test may be the only option

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

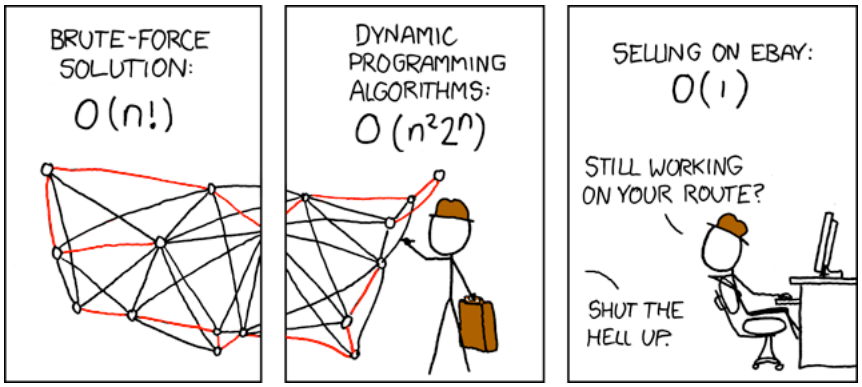
efficiency?



14

xkcd: Traveling Salesman Problem comic

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



15

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

16

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());
    }
}
```

17

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;
    }
}
```

18

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);
  }
}
```

19

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) { … }
    …
}
```

20

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);
    }
}
```

21

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);
    }
}
```

22

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?

23

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);
  }
}
```

24

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);
    }
}
```

25

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);
  }
}
```

26