CSE 143
Lecture 15

Recursive Backtracking

slides created by Marty Stepp
http://www.cs.washington.edu/143/
ideas and examples taken from Stanford University CS slides/lectures

Exercise: Permutations

- Write a method `permute` that accepts a string as a parameter and outputs all possible rearrangements of the letters in that string. The arrangements may be output in any order.
  - Example: `permute("MARTY")` outputs the following sequence of lines:

MARTY	MYRAT	ATYMR	RTMAY	TARMY	YMTAR
MARYT	MYRTA	ATYRM	RTMYA	TARYM	YMTRA
MATRY	MYTAR	AYMRT	RTAMY	TAYMR	YAMRT
MATYR	MYTRA	AYMTR	RTAYM	TAYRM	YAMTR
MAYRT	AMRTY	AYRMT	RTYMA	TRMAY	YARMT
MAYTR	AMRYT	AYRTM	RTYAM	TRMYA	YARTM
MRATY	AMTRY	AYTMR	RYMAT	TRAMY	YATMR
MRAYT	AMTYR	AYTRM	RYMTA	TRAYM	YATRM
MRTAY	AMYRT	RMATY	RYAMT	TRYMA	YRMAT
MRTYA	AMYTR	RMAYT	RYATM	TRYAM	YRMTA
MRYAT	ARMTY	RMTAY	RYTMA	TYMAR	YRAMT
MRYTA	ARMYT	RMTYA	RYTAM	TYMRA	YRATM
MTARY	ARTMY	RMYAT	TMARY	TYAMR	YRTMA
MTAYR	ARTYM	RMYTA	TMAYR	TYARM	YRTAM
MTRAY	ARYMT	RAMTY	TMRAY	TYRMA	YTMAR
MTRYA	ARYTM	RAMYT	TMRYA	TYRAM	YTMRA
MTYAR	ATMRY	RATMY	TMYAR	YMART	YTAMR
MTYRA	ATMYR	RATYM	TMYRA	YMATR	YTARM
MYART	ATRMY	RAYMT	TAMRY	YMRAT	YTRMA
MYATR	ATRYM	RAYTM	TAMYR	YMRTA	YTRAM

Examining the problem

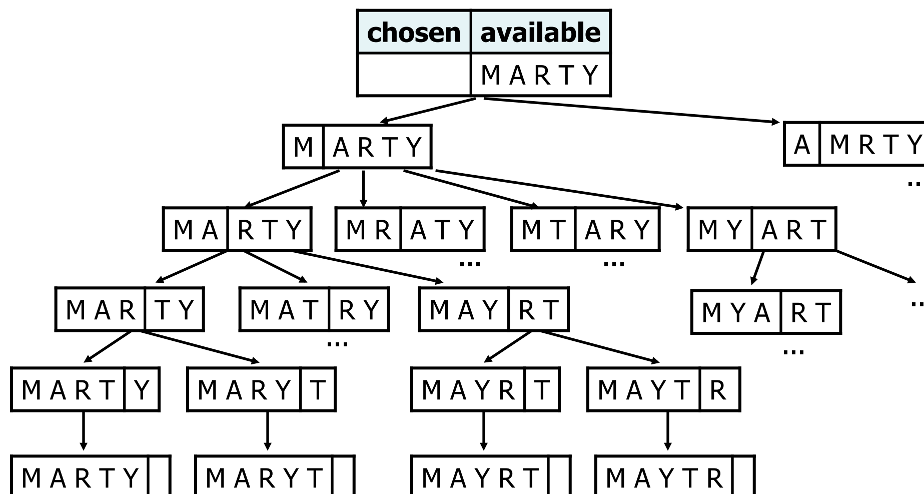
- Think of each permutation as a set of choices or **decisions**:
  - Which character do I want to place first?
  - Which character do I want to place second?
  - ...
  - **solution space**: set of all possible sets of decisions to explore

- We want to generate all possible sequences of decisions.

```
for (each possible first letter):
    for (each possible second letter):
        for (each possible third letter):
            ...
                print!
```

3

Decision trees



4

Backtracking

- **backtracking**: A general algorithm for finding solution(s) to a computational problem by trying partial solutions and then abandoning them ("backtracking") if they are not suitable.
  - a "brute force" algorithmic technique (tries all paths; not clever)
  - often (but not always) implemented recursively

  Applications:
  - producing all permutations of a set of values
  - parsing languages
  - games: anagrams, crosswords, word jumbles, 8 queens
  - path-planning (e.g., robots or parts assembly)

Backtracking algorithms

A general pseudo-code algorithm for backtracking problems:

explore(**choices**):

- if there are no more **choices** to make: stop.
- else:
  - Make a single choice **C** from the set of choices.
    - Remove **C** from the set of **choices**.
  - explore the remaining **choices**.
  - Un-make choice **C**.
    - Backtrack!

Backtracking strategies

- When solving a backtracking problem, ask these questions:
  - What are the "choices" in this problem?
    - What is the "base case"? (How do I know when I'm out of choices?)
  - How do I "make" a choice?
    - Do I need to create additional variables to remember my choices?
    - Do I need to modify the values of existing variables?
  - How do I explore the rest of the choices?
    - Do I need to remove the made choice from the list of choices?
  - Once I'm done exploring the rest, what should I do?
  - How do I "un-make" a choice?

Permutations revisited

- Write a method `permute` that accepts a string as a parameter and outputs all possible rearrangements of the letters in that string. The arrangements may be output in any order.
  - Example:
    `permute("MARTY")`
    outputs the following sequence of lines:

MARTY	MYRAT	ATYMR	RTMAY	TARMY	YMTAR
MARYT	MYRTA	ATYRM	RTMYA	TARYM	YMTRA
MATRY	MYTAR	AYMRT	RTAMY	TAYMR	YAMRT
MATYR	MYTRA	AYMTR	RTAYM	TAYRM	YAMTR
MAYRT	AMRTY	AYRMT	RTYMA	TRMAY	YARMT
MAYTR	AMRYT	AYRTM	RTYAM	TRMYA	YARTM
MRATY	AMTRY	AYTMR	RYMAT	TRAMY	YATMR
MRAYT	AMTYR	AYTRM	RYMTA	TRAYM	YATRM
MRTAY	AMYRT	RMATY	RYAMT	TRYMA	YRMAT
MRTYA	AMYTR	RMAYT	RYATM	TRYAM	YRMTA
MRYAT	ARMTY	RMTAY	RYTMA	TYMAR	YRAMT
MRYTA	ARMYT	RMTYA	RYTAM	TYMRA	YRATM
MTARY	ARTMY	RMYAT	TMARY	TYAMR	YRTMA
MTAYR	ARTYM	RMYTA	TMAYR	TYARM	YRTAM
MTRAY	ARYMT	RAMTY	TMRAY	TYRMA	YTMAR
MTRYA	ARYTM	RAMYT	TMRYA	TYRAM	YTMRA
MTYAR	ATMRY	RATMY	TMYAR	YMART	YTAMR
MTYRA	ATMYR	RATYM	TMYRA	YMATR	YTARM
MYART	ATRMY	RAYMT	TAMRY	YMRAT	YTRMA
MYATR	ATRYM	RAYTM	TAMYR	YMRTA	YTRAM

Exercise solution

```
// Outputs all permutations of the given string.
public static void permute(String s) {
    permute(s, "");
}

private static void permute(String s, String chosen) {
    if (s.length() == 0) {
        // base case: no choices left to be made
        System.out.println(chosen);
    } else {
        // recursive case: choose each possible next letter
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);                   // choose
            s = s.substring(0, i) + s.substring(i + 1);
            chosen += c;

            permute(s, chosen);                     // explore

            s = s.substring(0, i) + c + s.substring(i);
            chosen = chosen.substring(0, chosen.length() - 1);
        }                                           // un-choose
    }
}
```

9

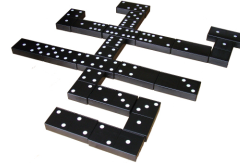
Exercise solution 2

```
// Outputs all permutations of the given string.
public static void permute(String s) {
    permute(s, "");
}

private static void permute(String s, String chosen) {
    if (s.length() == 0) {
        // base case: no choices left to be made
        System.out.println(chosen);
    } else {
        // recursive case: choose each possible next letter
        for (int i = 0; i < s.length(); i++) {
            char ch = s.charAt(i);                  // choose

            String rest = s.substring(0, i) +       // remove
                          s.substring(i + 1);

            permute(rest, chosen + ch);             // explore
        }
    }       // (don't need to "un-choose" because
}           //  we used temp variables)
```

10

Exercise: Dominoes

- The game of dominoes is played with small black tiles, each having 2 numbers of dots from 0-6. Players line up tiles to match dots.



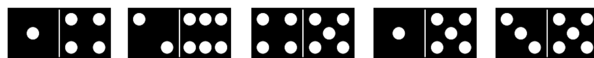
- Given a class `Domino` with the following methods:

```
public int first()          // first dots value
public int second()         // second dots value
public String toString()    // e.g. "(3|5)"
```

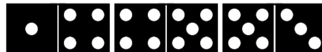
- Write a method `hasChain` that takes a `List` of dominoes and a starting/ending dot value, and returns whether the dominoes can be made into a chain that starts/ends with those values.
  - If the chain's start/end are the same, the answer is always `true`.

Domino chains

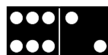
- Suppose we have the following dominoes:



- We can link them into a chain from 1 to 3 as follows:
  - Notice that the 3|5 domino had to be flipped.



- We can "link" one domino into a "chain" from 6 to 2 as follows:



Exercise client code

```
import java.util.*;    // for ArrayList

public class SolveDominoes {
    public static void main(String[] args) {
        // [(1|4), (2|6), (4|5), (1|5), (3|5)]
        List<Domino> dominoes = new ArrayList<Domino>();
        dominoes.add(new Domino(1, 4));
        dominoes.add(new Domino(2, 6));
        dominoes.add(new Domino(4, 5));
        dominoes.add(new Domino(1, 5));
        dominoes.add(new Domino(3, 5));
        System.out.println(hasChain(dominoes, 5, 5));    // true
        System.out.println(hasChain(dominoes, 1, 5));    // true
        System.out.println(hasChain(dominoes, 1, 3));    // true
        System.out.println(hasChain(dominoes, 1, 6));    // false
        System.out.println(hasChain(dominoes, 1, 2));    // false
    }

    public static boolean hasChain(List<Domino> dominoes,
                                   int start, int end) {
        ...
    }
}
```

13

Exercise solution

```
public static boolean hasChain(List<Domino> dominoes,
                               int start, int end) {
    if (start == end) {
        return true;                         // base case
    } else {
        for (int i = 0; i < dominoes.size(); i++) {
            Domino d = dominoes.remove(i);   // choose
            if (d.first() == start) {        // explore
                if (hasChain(dominoes, d.second(), end)) {
                    dominoes.add(i, d);      // un-choose
                    return true;
                }
            } else if (d.second() == start) {
                if (hasChain(dominoes, d.first(), end)) {
                    dominoes.add(i, d);      // un-choose
                    return true;
                }
            }
            dominoes.add(i, d);              // un-choose
        }
        return false;
    }
}
```

14

Exercise: Print chain

- Write a variation of your `hasChain` method that also prints the chain of dominoes that it finds, if any.

```
hasChain(dominoes, 1, 3);

[(1|4), (4|5), (5|3)]
```

15

Exercise solution

```
public static boolean hasChain(List<Domino> dominoes, int start, int end) {
    Stack<Domino> chosen = new ArrayStack<Domino>();
    return hasChain(dominoes, chosen, start, end);
}

private static boolean hasChain(List<Domino> dominoes,
                                Stack<Domino> chosen, int start, int end) {
    if (start == end) {
        System.out.println(chosen);
        return true;                          // base case
    } else {
        for (int i = 0; i < dominoes.size(); i++) {
            Domino d = dominoes.remove(i);    // choose
            if (d.first() == start) {         // explore
                chosen.push(d);
                if (hasChain(dominoes, chosen, d.second(), end)) {
                    dominoes.add(i, d);       // un-choose
                    return true;
                }
                chosen.pop();
            } else if (d.second() == start) {
                d.flip();
                chosen.push(d);
                if (hasChain(dominoes, chosen, d.second(), end)) {
                    dominoes.add(i, d);       // un-choose
                    return true;
                }
                d.flip();
                chosen.pop();
            }
            dominoes.add(i, d);               // un-choose
        }
        return false;
    }
}
```

16

Exercise: Print all chains

- Write a variation of your `hasChain` method called `findChains` that prints all possible chains of dominoes, if any.

```
findChains(dominoes, 1, 3);

[(1|4), (4|5), (5|3)]
[(1|5), (5|3)]
```

- As a variation, write the method so that if the start and end numbers are the same, the chain must have at least one domino (instead of always being true!)

17

Exercise solution

```
public static void findChains(List<Domino> dominoes, int start, int end) {
    Stack<Domino> chosen = new ArrayStack<Domino>();
    findChains(dominoes, chosen, start, end);
}

private static void findChains(List<Domino> dominoes,
                               Stack<Domino> chosen, int start, int end) {
    if (chosen.size() > 0 && start == end) {
        System.out.println(chosen);          // base case
    } else {
        for (int i = 0; i < dominoes.size(); i++) {
            Domino d = dominoes.remove(i);   // choose
            if (d.first() == start) {        // explore
                chosen.push(d);
                findChains(dominoes, chosen, d.second(), end);
                chosen.pop();
            } else if (d.second() == start) {
                d.flip();
                chosen.push(d);
                findChains(dominoes, chosen, d.second(), end);
                d.flip();
                chosen.pop();
            }
            dominoes.add(i, d);              // un-choose
        }
    }
}
```

18