csci 210: Data Structures

More Recursion
Summary

• Topics: more recursion

  • Subset sum: finding if a subset of an array that sum up to a given target
  • Permute: finding all permutations of a given string
  • Subset: finding all subsets of a given string
Thinking recursively

• Finding the recursive structure of the problem is the hard part

• Common patterns
  • divide in half, solve one half
  • divide in sub-problems, solve each sub-problem recursively, “merge”
  • solve one or several problems of size n-1
  • process first element, recurse on remaining problem

• Recursion
  • functional: function computes and returns result.
    • Example: computing the sum of n numbers; isPalindrome; binary search.
  • procedural: no return result (function returns void). The task is accomplished during the recursive calls.
    • Example: Sierpinski fractals.

• Recursion
  • exhaustive
  • non-exhaustive: stops early
Subset Sum

• Given an array of numbers and a target value, find whether there exists a subset of those numbers that sum up to the target value.

boolean subsetSum (int[] a, int target)

• Example:

• Recursive structure:
  • consider the next element in the array
  • try making a sum WITH this element
  • try making a sum WITHOUT this element
  • if neither is possible, return false
Subset Sum

- So: consider the next element, it is either in the solution, or not. Try both ways. If both fail, return false.
- Need to keep track of the partial sum so far. When starting a recursive call, need to know the sum of the current subset. Also need to know the index of the next element to consider.

```java
void recSubset(int[] a, int target, int i, int sumSoFar)
```

- The problem asked for a subsetSum function with the following signature:
  ```java
  boolean subsetSum (int[] a, int target)
  ```

- Need a wrapper:
  ```java
  boolean subsetSum (int[] a, int target) {
      return recSubset(a, target, 0, 0);
  }
  ```
boolean recSubset(int[] a, int target, int i, int sumSoFar) {
    // base cases
    // we got it
    if (sumSoFar == target) return true;
    // we reached the end and sum is not equal to target
    if (i == a.length) return false;

    // recursive case: try next element both in and out of the sum
    boolean with = recSubset(a, target, i+1, sumSoFar + a[i]);
    boolean without = recSubset(a, target, i+1, sumSoFar);
    return (with || without);
}
• The tree of recursive calls for recSubset([1, 2, 3, 4], target, 0, 0)
Subset Sum

• Variations
  • Alternative strategy: at each step, chose one of the remaining element to be part of the subset and recurse on the remaining part.

  • How could you change the function so that it prints the elements of the subset that sum to target?
    • store partial subsets in another array
    • or print element at the end of recursive call

  • How could you change the function to report not only if such a subset exists, but to count all such subsets?
Permutations

• Write a function to print all permutations of a given string.
• Example: permute “abc” should print: abc, acb, bca, bac, cab, cba.

void printPerm(String s)

• Recursive structure:
  • Chose a letter from the input, and make this the first letter of the output
  • Recursively permute remaining input
  • chose a, permute “bc”: should generate “a” + all permutations of “bc”
  • chose all letters in turn to be first letters
  • chose b, permute “ac”: should generate “b” + all permutations of “ac”
  • chose c, permute “ab”: should generate “c” + all permutations of “ab”

• What is the base case?
• Can you make sure that each permutation is generated precisely once?
Permutations

• So: pick a letter, add it to the solution, recurse on remaining
• When starting a recursive call, we know the list of letters chosen so far; that is, we know the first part of the permutation generated so far.
• Need to keep track of it.

//print soFar + all permutations of remaining
void recPermute(String soFar, String remaining)

• The problem asked for a printPermute with a different signature: we need a wrapper
  //print all permutations of s
  void printPerm (String s) {
    recPermute("", s);
  }

• Why use wrappers? the user does not need to know the internals of the
void recPermute(String soFar, String remaining) {

    //base case
    if (remaining.length() == 0)
        System.out.println(soFar);
    else {
        for (int i=0; i< remaining.length(); i++) {
            String nextSoFar = soFar + remaining[i];
            String nextRemaining = remaining.substring(0,i) +  remaining.substring(i+1);
            recPermute(nextSoFar, nextRemaining)
        }
    }
}

Permutations

- The tree of recursive calls for recPermute("", "abc")
• Enumerate all subsets of a given string
  • Example: subsets of “abc” are a, b, c, ab, ac, bc, abc
    • Order does not matter: “ab” is the same as “ba”

• Recursive structure
  • chose one element from input
  • can either include it in current subset or not
  • recursively form subsets including it
  • recursively form subsets excluding it
  • make sure to generate each set once
  • base case?
void recSubsets(String soFar, String remaining) {
    if (remaining.length()==0)
        System.out.println(soFar);
    else {
        //add to subset, remove from rest, recurse
        recSubsets(soFar+remaining[0], remaining.substring(1);
        //don’t add to subset, remove from rest, recurse
        recSubsets(soFar, remaining.substring(1);
    }
}

void subsets(String s) {
    recSubsets(““, s);
}
Subsets

- The tree of recursive calls for recSubsets("", "abcd")