Recursion in CS 104

• Problem in which the solution can be expressed in terms of itself (usually a smaller instance/input of the same problem) and a base/terminating case

• Recursion is a key concept in this course
  – But it rarely comes easily to students. You must work at it!

• Many problems that would be VERY difficult to solve without recursion (i.e. only loops) have extremely elegant solutions to problems
  – Learn to look for those elegant solutions
  – In this class, assume the recursive approach has an elegant/simple solution
  – If you find yourself writing a large, complex recursive solution, assume you are doing something you should not!
    • Stop and reconsider how it should be done
Simple vs. Multiple Recursion

- "Simple" recursion refers to functions that contain just **ONE recursive call**
  - Can be head or tail recursion (explained soon)
  - Can easily be replaced by a loop

- The power of recursion usually comes when the function makes **2 OR MORE recursive calls** (aka "multiple recursion")
  - Elegant recursive solutions that would be **MUCH harder to implement iteratively** (usually need a separate stack data structure)

- We'll focus on **multiple recursion**

```cpp
void print(Item* p) {
    if(p == NULL) return;
    else {
        cout << p->val << endl;
        print(p->next);
    }
}
```

**Simple Recursion** (1 recursive call)

```cpp
void postorder(TNode* t) {
    if(t == NULL) return
    postorder(t->left)
    postorder(t->right)
    process(t) // print val.
}
```

**Multiple Recursion** (2 or more recursive calls)
Steps to Formulating Recursive Solutions

1. Solve a few instances of the problem to discover the recursive structure

2. Identify how the problem can be decomposed into smaller problems of the same form
   – Does solving the problem on an input of smaller value or size help formulate the solution to the larger

3. Identify the base case
   – An input for which the answer is trivial

4. Assume the recursive call for the smaller problem "magically" computes the correct solution(s) to those problem(s) and **identify how to combine those solution(s)** from the smaller problem(s) into the solution for the larger problem
Towers of Hanoi Problem

- Problem Statements: Move n discs from source pole to destination pole (with help of a 3rd alternate pole)
  - Cannot place a larger disc on top of a smaller disc
  - Can only move one disc at a time

Start (n=3) → Goal (n=3)

Not allowed
Observation 1

- Observation 1: Disc 1 (smallest) can always be moved
- Solve the n=2 case:

Start

Move 1 from src to alt

Move 2 from src to dst

Move 1 from alt to dst
Observation 2

- Observation 2: If there is only one disc on the src pole and the dest pole can receive it the problem is trivial

```
A (src)  B (dst)  C (alt)

3

Move n-1 discs from src to alt

Move disc n from src to dst

Move n-1 discs from alt to dst
```
Recursive solution

• But to move $n-1$ discs from $src$ to $alt$ is really a smaller version of the same problem with
  - $n => n-1$
  - $src => src$
  - $alt => dst$
  - $dst => alt$

• $Towers(n, src, dst, alt)$
  - Base Case: $n == 1$  // Observation 1: Disc 1 always movable
    • Move disc 1 from $src$ to $dst$
  - Recursive Case:  // Observation 2: Move of $n-1$ discs to $alt$ & back
    • $Towers(n-1, src, alt, dst)$
    • Move disc $n$ from $src$ to $dst$
    • $Towers(n-1, alt, dst, src)$
Recursive Box Diagram

Towers Function Prototype

Towers(disc,src,dst,alt)

Towers(3,a,b,c)
  - Move D=3 a to b
  - Towers(2,c,b,a)
    - Move D=2 c to b
    - Towers(1,c,a,b)
      - Move D=1 c to a
    - Towers(1,a,b,c)
      - Move D=1 a to b
  - Towers(2,a,c,b)
    - Move D=2 a to c
    - Towers(1,b,c,a)
      - Move D=1 b to c
  - Towers(1,a,b,c)
    - Move D=1 a to b
GENERATING ALL COMBINATIONS
Recursion's Power

• The power of recursion often comes when each function instance makes *multiple* recursive calls

• As you will see this often leads to an exponential number of "combinations" being generated/explored in an easy fashion
Binary Combinations

• If you are given the value, n, and a string with n characters could you generate all the combinations of n-bit binary?

• Do so recursively!

Exercise: bin_combo_str
Recursion and DFS

- Recursion forms a kind of Depth-First Search

```
// user interface
void binCombos(int len)
{
    binCombos("", len);
}

// helper-function
void binCombos(string prefix, int len)
{
    if(prefix.length() == len )
        cout << prefix << endl;
    else {
        // recurse
        binCombos(___________, len);
        // recurse
        binCombos(___________, len);
    }
}
```

Generally: Recursion must perform the same code sequence for each item. Where we need variation, use 'if' statements.
Generating All Combinations

• Recursion offers a simple way to generate all \(N\)-length combinations of from a set of options, \(S\)
  
  – Example: Generate all 2-digit decimal numbers (\(N=2, S=\{0,1,...,9\}\))

```cpp
void NDigDecCombos(string data, int n)
{
    if(data.size() == n )
        cout << data;
    else {
        for(int i=0; i < 10; i++){
            // recurse
            NDigDecCombos(data+(char)('0'+i),n);
        }
    }
}
```

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.
Another Exercise

• Generate all string combinations of length n from a given list (vector) of characters

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

void all_combos(vector<char>& letters, int n) {
    // ???
}

int main() {
    vector<char> letters = {'U', 'S', 'C'};

    all_combos(letters, 4);
    return 0;
}
```

Use recursion to walk down the 'places'
At each 'place' iterate through & try all options
Recursion and Combinations

• Recursion provides an elegant way of generating all $n$-length combinations of a set of values, $S$.
  – Ex. Generate all length-$n$ combinations of the letters in the set $S=${'U','S','C'} (i.e. for n=2: UU, US, UC, SU, SS, SC, CU, CS, CC)

• General approach:
  – Need some kind of array/vector/string to store partial answer as it is being built
  – Each recursive call is only responsible for one of the $n$ "places" (say location, $i$)
  – The function will iteratively (loop) try each option in $S$ by setting location $i$ to the current option, then recurse to handle all remaining locations ($i+1$ to $n$)
    • Remember you are responsible for only one location
  – Upon return, try another option value and recurse again
  – Base case can stop when all $n$ locations are set (i.e. recurse off the end)
  – Recursive case returns after trying all options
Exercises

• bin_combos_str
• Zero_sum
• Prime_products_print
• Prime_products
• basen_combos
• all_letter_combos
Recursive Backtracking Search

• Recursion allows us to "easily" enumerate all solutions/combinations to some problem
• Backtracking algorithms are often used to solve constraint satisfaction problems or optimization problems
  – Find (the best) solutions/combinations that meet some constraints
• **Key property of backtracking search:**
  – Stop searching down a path at the first indication that constraints won't lead to a solution
• Many common and important problems can be solved with backtracking approaches
• Knapsack problem
  – You have a set of products with a given weight and value. Suppose you have a knapsack (suitcase) that can hold N pounds, which subset of objects can you pack that maximizes the value.
  – Example:
    • Knapsack can hold 35 pounds
    • Product A: 7 pounds, $12 ea.  Product B: 10 pounds, $18 ea.
    • Product C: 4 pounds, $7 ea.  Product D: 2.4 pounds, $4 ea.
• Other examples:
  – Map Coloring, Satisfiability, Sudoku, N-Queens

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.
N-Queens Problem

- Problem: How to place N queens on an NxN chess board such that no queens may attack each other
- Fact: Queens can attack at any distance vertically, horizontally, or diagonally
- Observation: Different queen in each row and each column
- Backtrack search approach:
  - Place 1\textsuperscript{st} queen in a viable option then, then try to place 2\textsuperscript{nd} queen, etc.
  - If we reach a point where no queen can be placed in row i or we've exhausted all options in row i, then we return and change row i-1
8x8 Example of N-Queens

• Now place 2\textsuperscript{nd} queen
8x8 Example of N-Queens

• Now place others as viable
• After this configuration here, there are no locations in row 6 that are not under attack from the previous 5
• BACKTRACK!!!
8x8 Example of N-Queens

- Now place others as viable
- After this configuration here, there are no locations in row 6 that is not under attack from the previous 5
- So go back to row 5 and switch assignment to next viable option and progress back to row 6
8x8 Example of N-Queens

- Now place others as viable
- After this configuration here, there are no locations in row 6 that is not under attack from the previous 5
- Now go back to row 5 and switch assignment to next viable option and progress back to row 6
- But still no location available so return back to row 5
8x8 Example of N-Queens

- Now place others as viable
- After this configuration here, there are no locations in row 6 that is not under attack from the previous 5
- Now go back to row 5 and switch assignment to next viable option and progress back to row 6
- But still no location available so return back to row 5
- But now no more options for row 5 so return back to row 4
- BACKTRACK!!!!
8x8 Example of N-Queens

• Now place others as viable
• After this configuration here, there are no locations in row 6 that is not under attack from the previous 5
• Now go back to row 5 and switch assignment to next viable option and progress back to row 6
• But still no location available so return back to row 5
• But now no more options for row 5 so return back to row 4
• Move to another place in row 4 and restart row 5 exploration
8x8 Example of N-Queens

• Now place others as viable
• After this configuration here, there are no locations in row 6 that is not under attack from the previous 5
• Now go back to row 5 and switch assignment to next viable option and progress back to row 6
• But still no location available so return back to row 5
• But now no more options for row 5 so return back to row 4
• Move to another place in row 4 and restart row 5 exploration
8x8 Example of N-Queens

- Now a viable option exists for row 6
- Keep going until you successfully place row 8 in which case you can return your solution
- What if no solution exists?
8x8 Example of N-Queens

• Now a viable option exists for row 6
• Keep going until you successfully place row 8 in which case you can return your solution
• What if no solution exists?
  – Row 1 queen would have exhausted all her options and still not find a solution
Backtracking Search

• Recursion can be used to generate all options
  – 'brute force' / test all options approach
  – Test for constraint satisfaction only at the bottom of the 'tree'

• But backtrack search attempts to 'prune' the search space
  – Rule out options at the partial assignment level

Brute force enumeration might test only when a complete assignment is made (i.e. all 4 queens on the board)
N-Queens Solution Development

• Let's develop the code
  • 1 queen per row
    – Use an array where index represents the queen (and the row) and value is the column
  • Start at row 0 and initiate the search [i.e. search(0) ]
  • Base case:
    – Rows range from 0 to n-1 so STOP when row == n
    – Means we found a solution
  • Recursive case
    – Recursively try all column options for that queen
    – But haven't implemented check of viable configuration...

```c
int *q;  // pointer to array storing
         // each queens location
int n;   // number of board / size

void search(int row)
{
    if(row == n)
        printSolution(); // solved!
    else {
        // remember q[row] is the column
        for(q[row]=0; q[row]<n; q[row]++){
            search(row+1);
        }
        // alternatively
        // for(int col = 0; col < n; col++){
        //     q[row] = col;
        //     search(row+1);
        // }
```
N-Queens Solution Development

• To check whether it is safe to place a queen in a particular column, let's keep a "threat" 2-D array indicating the threat level at each square on the board
  – Threat level of 0 means SAFE
  – When we place a queen we'll update squares that are now under threat
  – Let's name the array 't'

• Dynamically allocating 2D arrays in C/C++ doesn't really work
  – Instead conceive of 2D array as an "array of arrays" which boils down to a pointer to a pointer

```
int *q; // pointer to array storing
   // each queens location
int n; // number of board / size
int **t; // thread 2D array

int main()
{
    q = new int[n];
    t = new int*[n];
    for(int i=0; i < n; i++){
        t[i] = new int[n];
        for(int j = 0; j < n; j++){
            t[i][j] = 0;
        }
    }
    search(0); // start search
    // deallocate arrays
    return 0;
}
```
N-Queens Solution Development

- After we place a queen in a location, let's check that it has no threats
- If it's safe then we update the threats (+1) due to this new queen placement
- Now recurse to next row
- If we return, it means the problem was either solved or more often, that no solution existed given our placement so we remove the threats (-1)
- Then we iterate to try the next location for this queen

```c
int *q;  // pointer to array storing each queens location
int n;   // number of board / size
int **t; // n x n threat array
void search(int row)
{
    if(row == n)
        printSolution(); // solved!
    else {
        for(q[row]=0; q[row]<n; q[row]++){
            // check that col: q[row] is safe
            if(t[row][q[row]] == 0){
                // if safe place and continue
                addToThreats(row, q[row], 1);
                search(row+1);
                // if return, remove placement
                addToThreats(row, q[row], -1);
            }
        }
    }
}
```
addToThreats Code

• Observations
  – Already a queen in every higher row so addToThreats only needs to deal with positions lower on the board
    • Iterate row+1 to n-1
  – Enumerate all locations further down in the same column, left diagonal and right diagonal
  – Can use same code to add or remove a threat by passing in change

• Can't just use 2D array of booleans as a square might be under threat from two places and if we remove 1 piece we want to make sure we still maintain the threat

```java
void addToThreats(int row, int col, int change) {
    for(int j = row+1; j < n; j++){
        // go down column
        t[j][col] += change;
        // go down right diagonal
        if( col+(j-row) < n )
            t[j][col+(j-row)] += change;
        // go down left diagonal
        if( col-(j-row) >= 0)
            t[j][col-(j-row)] += change;
    }
}
```
N-Queens Solution

```c++
int *q; // queen location array
int n; // number of board / size
int **t; // n x n threat array

int main()
{
    q = new int[n];
t = new int*[n];
for(int i=0; i < n; i++)
    t[i] = new int[n];
for(int j = 0; j < n; j++)
    t[i][j] = 0;

// do search
if( !search(0) )
    cout << "No sol!" << endl;
// deallocate arrays
return 0;
}

void addToThreats(int row, int col, int change)
{
    for(int j = row+1; j < n; j++)
    {
        // go down column
        t[j][col] += change;
        // go down right diagonal
        if( col+(j-row) < n )
            t[j][col+(j-row)] += change;
        // go down left diagonal
        if( col-(j-row) >= 0 )
            t[j][col-(j-row)] += change;
    }
}

bool search(int row)
{
    if(row == n)
    {
        printSolution(); // solved!
        return true;
    }
    else {
        for(q[row]=0; q[row]<n; q[row]++)
        {
            // check that col: q[row] is safe
            if(t[row][q[row]] == 0)
            {
                // if safe place and continue
                addToThreats(row, q[row], 1);
                bool status = search(row+1);
                if(status) return true;
                // if return, remove placement
                addToThreats(row, q[row], -1);
            }
        }
        return false;
    }
}
```

General Backtrack Search Approach

- Select an item and set it to one of its options such that it meets current constraints
- Recursively try to set next item
- If you reach a point where all items are assigned and meet constraints, done...return through recursion stack with solution
- If no viable value for an item exists, backtrack to previous item and repeat from the top
- If viable options for the 1st item are exhausted, no solution exists
- Phrase:
  - Assign, recurse, unassign

```c
bool sudoku(int **grid, int r, int c)
{
    if( allSquaresComplete(grid) )
        return true;
    // iterate through all options
    for(int i=1; i <= 9; i++){
        grid[r][c] = i;
        if( isValid(grid) ){
            bool status = sudoku(...);
            if(status) return true;
        }
    }
    return false;
}
```

Assume r,c is current square to set and grid is the 2D array of values
Runtime of All Combinations

- \( T(n_r, n_c) = \) _________________
- \( T(0, n_c) = 1 \)

```c
int *q; // pointer to array storing
         // each queens location
int n;   // number of board / size

void search(int row)
{
    if(row == n)
        printSolution(); // solved!
    else {
        // remember q[row] is the column
        for(q[row]=0; q[row]<n; q[row]++){
            search(row+1);
        }
    }
}
```
SOLUTIONS
Recursion and DFS

- Recursion forms a kind of Depth-First Search

Generally: Recursion must perform the same code sequence for each item. Where we need variation, use 'if' statements.

// user interface
void binCombos(int len)
{
    binCombs("", len);
}

// helper-function
void binCombos(string prefix, int len)
{
    if (prefix.length() == len)
        cout << prefix << endl;
    else {
        // recurse
        binCombos(prefix + "0", len);
        // recurse
        binCombos(prefix + "1", len);
    }
}