

CSCI-1200 Computer Science II — Spring 2006

Lecture 10

Recursion Examples

Review from Lecture 9

- Recursive definitions and C++ functions
- The mechanism of recursion using activation records
- Recursion vs. iteration
- Rules for writing recursive functions
 1. Handle the base case(s)
 2. Define the problem in terms of smaller problems
 3. What happens before the recursive call(s)
 4. What happens after the recursive call(s)
 5. Make sure you are proceeding toward the base case(s)
- Examples:
 - Printing the contents of the vector in reverse order
 - Binary search
- Writing *templated functions!*

Today's Class

- Finish discussion of binary search
- Two examples that are hard to solve without using recursion:
 - Merge sort
 - Nonlinear word search

10.1 Binary Search Review: What's wrong with this code?

```
bool binsearch(vector<double>& v, double x) {  
    return binsearch(v, x, 0, v.size()-1);  
}
```

```
bool binsearch(vector<double>& v, double x, int low, int high) {  
    if (low == high)  
        return x == v[low];  
    int mid = (low+high)/2;  
    if (x >= v[mid])  
        return binsearch(v, x, mid, high);  
    else  
        return binsearch(v, x, low, mid-1);  
}
```

10.2 Recursion Example: Merge Sort

- Idea:
 - Split a vector in half
 - Recursively sort each half
 - Merge the two sorted halves into a single sorted vector
- Suppose we have a vector called `values` having two halves that are each already sorted. In particular, the values in subscript ranges `[low..mid]` (the lower interval) and `[mid+1..high]` (the upper interval) are each in increasing order.
- Which values are candidates to be the first in the final sorted vector? Which values are candidates to be the second?
- In a loop, the merging algorithm repeatedly chooses one value to copy to `scratch`. At each step, there are only two possibilities: the first uncopied value from the lower interval and the first uncopied value from the upper interval.
- The copying ends when one of the two intervals is exhausted. Then the remainder of the other interval is copied into the scratch vector. Finally, the entire scratch vector is copied back.

10.3 Exercise: Complete the Merge Sort Implementation

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

template <class T> void mergesort(vector<T>& values);
template <class T> void mergesort(int low, int high, vector<T>& values, vector<T>& scratch);
template <class T> void merge(int low, int mid, int high, vector<T>& values, vector<T>& scratch);

int main() {
    vector<double> pts(7);
    pts[0] = -45.0; pts[1] = 89.0; pts[2] = 34.7; pts[3] = 21.1;
    pts[4] = 5.0; pts[5] = -19.0; pts[6] = -100.3;

    mergesort(pts);

    for (unsigned int i=0; i<pts.size(); ++i)
        cout << i << ": " << pts[i] << endl;
}

// The driver function for mergesort. It defines a scratch vector for temporary copies.
template <class T>
void mergesort(vector<T>& values) {
    vector<T> scratch(values.size());
    mergesort(0, int(values.size()-1), values, scratch);
}

// Here's the actual merge sort function. It splits the vector in
// half, recursively sorts each half, and then merges the two sorted
// halves into a single sorted interval.
template <class T>
void mergesort(int low, int high, vector<T>& values, vector<T>& scratch) {
    cout << "mergesort: low = " << low << ", high = " << high << endl;
    if (low >= high) // intervals of size 0 or 1 are already sorted!
        return;

    int mid = (low + high) / 2;
    mergesort(low, mid, values, scratch);
    mergesort(mid+1, high, values, scratch);
    merge(low, mid, high, values, scratch);
}
```

```

// Non-recursive function to merge two sorted intervals (low..mid & mid+1..high)
// of a vector, using "scratch" as temporary copying space.
template <class T>
void merge(int low, int mid, int high, vector<T>& values, vector<T>& scratch) {
    cout << "merge: low = " << low << ", mid = " << mid << ", high = " << high << endl;
    int i=low, j=mid+1, k=low;

}

```

10.4 Thinking About Merge Sort

- It exploits the power of recursion! We only need to think about
 - Base case (intervals of size 1)
 - Splitting the vector
 - Merging the results
- We will insert `cout` statements into the algorithm and use this to understand how this is is happening.
- Can we analyze this algorithm and determine the order notation for the number of operations it will perform? Count the number of pairwise comparisons that are required.

10.5 Example: Word Search

- Take a look at the following grid of characters.

```

heanfuyaadfj
crarneradfad
chenenssartr
kdfthileerdr
chadufjavcze
dfhoepradlfc
neicpemrtlkf
paermerohtrr
diofetaycrhg
daldruetryrt

```

- The usual problem associated with a grid like this is to find words going forward, backward, up, down, or along a diagonal. Can you find “computer”?

- A sketch of the solution is as follows:
 - The grid of letters is represented as `vector<string> grid`; Each string represents a row. We can treat this as a *two-dimensional array*.
 - A word to be sought, such as “computer” is read as a string.
 - A pair of nested for loops searches the grid for occurrences of the first letter in the string. Call such a location (r, c)
 - At each such location, the occurrences of the second letter are sought in the 8 locations surrounding (r, c) .
 - At each location where the second letter is found, a search is initiated in the direction indicated. For example, if the second letter is at $(r, c - 1)$, the search for the remaining letters proceeds up the grid.
- The implementation takes a bit of work, but is not too bad. It’s similar to the exercises from lab earlier this week and we could have asked you to do this problem instead.

10.6 Example: Nonlinear Word Search

- Today we’ll work on a different, but somewhat harder problem: What happens when we no longer require the locations to be along the same row, column or diagonal of the grid, but instead allow the locations to snake through the grid? The only requirements are that
 1. the locations of adjacent letters are connected along the same row, column or diagonal, and
 2. a location can not be used more than once in each word
- Can you find `rensselaer`? It is there. How about `temperature`? Close, but nope!
- The implementation of this is very similar to the implementation described above until after the first letter of a word is found.
- We will look at the code during lecture, and then consider how to write the recursive function.

10.7 Exercise: Complete the implementation

```
// Program: word_search
// Author:  Chuck Stewart
//
// Purpose: A program to solve the word search problem where the
// letters in the word do not need to appear along a straight
// line. Instead, they can twist and turn. The only requirements are
// two-fold: the consecutive letters must be "8-connected" to each
// other (meaning that the locations must touch along an edge or at a
// corner), and no location may be used more than once.
//
// The real issue is how to search for the letters and then record
// locations as we search. This is most easily done with a recursive
// function. This function will be written during lecture.
//
// The input is from an input file. The grid is a series of strings,
// ended by a string that begins with '-'. Each subsequent string in
// the file is used to search the input.

#include <algorithm>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

// Simple class to record the grid location.
class loc {
public:
    loc(int r=0, int c=0) : row(r), col(c) {}
    int row, col;
};
```

```

bool operator== (const loc& lhs, const loc& rhs) {
    return lhs.row == rhs.row && lhs.col == rhs.col;
}

// Prototype for the main search function
bool search_from_loc(loc position, const vector<string>& board, const string& word, vector<loc>& path);

// Read in the letter grid, the words to search and print the results
int main(int argc, char* argv[]) {
    if (argc != 2) {
        cerr << "Usage: " << argv[0] << " grid-file\n";
        return 1;
    }
    ifstream istr(argv[1]);
    if (!istr) {
        cerr << "Couldn't open " << argv[1] << '\n';
        return 1;
    }

    vector<string> board;
    string word;
    vector<loc> path;          // The sequence of locations...
    string line;

    // Input of grid from a file. Stops when character '-' is reached.
    while ((istr >> line) && line[0] != '-')
        board.push_back(line);

    while (istr >> word) {
        bool found = false;
        vector<loc> path; // Path of locations in finding the word

        // Check all grid locations. For any that have the first
        // letter of the word, call the function search_from_loc
        // to check if the rest of the word is there.

        for (unsigned int r=0; r<board.size() && !found; ++r)
            for (unsigned int c=0; c<board[r].size() && !found; ++c) {
                if (board[r][c] == word[0] &&
                    search_from_loc(loc(r,c), board, word, path))
                    found = true;
            }

        // Output results
        cout << "\n** " << word << " ** ";
        if (found) {
            cout << "was found. The path is \n";
            for(unsigned int i=0; i<path.size(); ++i)
                cout << " " << word[i] << ": (" << path[i].row << ", " << path[i].col << ")\n";
        } else {
            cout << " was not found\n";
        }
    }
    return 0;
}

// helper function to check if a position has already been used for this word
bool on_path(loc position, vector<loc> const& path) {
    for (unsigned int i=0; i<path.size(); ++i)
        if (position == path[i]) return true;
    return false;
}

```

```

bool search_from_loc(loc position, // current position
    const vector<string>& board,
    const string& word,
    vector<loc>& path) // path up to the current pos
{

}

```

10.8 Summary of Nonlinear Word Search Recursion

- Recursion starts at each location where the first letter is found
- Each recursive call attempts to find the next letter by searching around the current position. When it is found, a recursive call is made.
- The current path is maintained at all steps of the recursion.
- The “base case” occurs when the path is full **or** all positions around the current position have been tried.

10.9 Exercise: Analyzing our Nonlinear Word Search Algorithm

What is the order notation for the number of operations?

Final Note

We’ve said that recursion is sometimes the *most natural way* to begin thinking about designing and implementing many algorithms. It’s ok if this feels downright uncomfortable right now. Practice, practice, practice!