CSCI-1200 Data Structures — Spring 2019
Lecture 12 — Advanced Recursion

Announcements: Test 2 Information

- Test 2 will be held **Thursday, February 28th, 2019 from 6-7:50pm**
  - Students will be randomly assigned to a test room and seating zone. If on Tuesday you still don't have a seating assignment when you log onto Submitty, let us know via the `ds_instructors` list.
  - No make-ups will be given except for pre-approved absence or illness, and a written excuse from the Dean of Students or the Student Experience office or the RPI Health Center will be required.
  - If you have a letter from Disability Services for Students and you have not already emailed it to `ds_instructors@cs.rpi.edu`, please do so by Tuesday 4:30pm. Erica Eberwein will be in contact with you about your accommodations for the test.
- Coverage: Lectures 1-13, Labs 1-7, HW 1-5.
- OPTIONAL: Prepare a 2 page, black & white, 8.5x11", portrait orientation .pdf of notes you would like to have during the exam. This may be digitally prepared or handwritten and scanned or photographed. The file may be no bigger than 2MB. You will upload this file to Submitty ("Test 2 Notes Upload") before Wednesday night @11:59pm. We will print this and attach it to your test.
- All students must bring their Rensselaer photo ID card.
- Practice problems from previous tests are available on the course website (as of Friday 11am). Solutions to the problems will be posted Monday at 10am.

Test Taking Skills

- Look at the point values for each problem, allocate time proportional to the problem points. (Don't spend all of your time on one problem and neglect other big point problems).
- Look at the size of the answer box & the sample solution code line estimate for each problem. If your solution is going to take a lot more space than the box allows, we are probably looking for the solution to a simpler problem or a simpler solution to the problem.
- Going in to the test, you should know what big topics will be covered on the test. As you skim through the problems, see if you can match up those big topics to each question. Even if you are stumped about how to solve the whole problem, or some of the details of the problem, make sure you demonstrate your understanding of the big topic that is covered in that question.
- Re-read the problem statement carefully. Make sure you didn't miss anything.
- Ask questions during the test if something is unclear.

Review from Lecture 11 & Lab 6

- Our own version of the STL `list<T>` class, named `dslist`
- Implementing `list<T>::iterator`
- Importance of destructors & using Dr. Memory / Valgrind to find memory errors
- Decrementing the `end()` iterator

Today's Lecture

- Review Recursion vs. Iteration
  - Binary Search
- "Rules" for writing recursive functions
- Advanced Recursion — problems that cannot be easily solved using iteration (for or while loops):
  - Merge sort
  - Non-linear maze search

12.1 Review: Iteration vs. Recursion

- Every recursive function can also be written iteratively. Sometimes the rewrite is quite simple and straightforward. Sometimes it's more work.
- Often writing recursive functions is more natural than writing iterative functions, especially for a first draft of a problem implementation.
- You should learn how to recognize whether an implementation is recursive or iterative, and practice rewriting one version as the other.
- Note: The order notation for the number of operations for the recursive and iterative versions of an algorithm is usually the same.

  However in C, C++, Java, and some other languages, *iterative functions are generally faster than their corresponding recursive functions.* This is due to the overhead of the function call mechanism.

  Compiler optimizations will sometimes (but not always!) reduce the performance hit by automatically eliminating the recursive function calls. This is called *tail call optimization.*

12.2 Binary Search

- Suppose you have a `std::vector<T> v` (for a placeholder type T), sorted so that:

  ```
  v[0] <= v[1] <= v[2] <= ...
  ```

- Now suppose that you want to find if a particular value `x` is in the vector somewhere. How can you do this without looking at every value in the vector?
- The solution is a recursive algorithm called *binary search*, based on the idea of checking the middle item of the search interval within the vector and then looking either in the lower half or the upper half of the vector, depending on the result of the comparison.

```
template <class T>
bool binsearch(const std::vector<T> &v, int low, int high, const T &x) {
 if (high == low) return x == v[low];
  int mid = (low+high) / 2;
  if (x <= v[mid])
    return binsearch(v, low, mid, x);
  else
    return binsearch(v, mid+1, high, x);
}
template <class T>
bool binsearch(const std::vector<T> &v, const T &x) {
  return binsearch(v, 0, v.size()-1, x);
}
```

12.3 Exercises

1. What is the order notation of binary search?

2. Write a non-recursive version of binary search.

3. If we replaced the if-else structure inside the recursive binsearch function (above) with

```
if ( x < v[mid] )
  return binsearch( v, low, mid-1, x );
else
  return binsearch( v, mid, high, x );
```

   would the function still work correctly?

12.4 "Rules" for Writing Recursive Functions

Here is an outline of five steps that are useful in writing and debugging recursive functions. Note: You don't have to do them in exactly this order...

1. Handle the base case(s).
2. Define the problem solution in terms of smaller instances of the problem. Use *wishful thinking*, i.e., if someone else solves the problem of `fact(4)` I can extend that solution to solve `fact(5)`. This defines the necessary recursive calls. It is also the hardest part!
3. Figure out what work needs to be done before making the recursive call(s).
4. Figure out what work needs to be done after the recursive call(s) complete(s) to finish the computation. (What are you going to do with the result of the recursive call?)
5. Assume the recursive calls work correctly, but make sure they are progressing toward the base case(s)!

12.5 Another Recursion Example: Merge Sort

- Idea: 1) Split a vector in half, 2) Recursively sort each half, and 3) 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.

12.6 Exercise: Complete the Merge Sort Implementation

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

int main() {
  std::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)
    std::cout << i << ": " << pts[i] << std::endl;
}

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

// Here's the actual merge sort function.  It splits the std::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, std::vector<T>& values, std::vector<T>& scratch) {
  std::cout << "mergesort:  low = " << low << ", high = " << high << std::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 std::vector, using "scratch" as temporary copying space.
template <class T>  void merge(int low, int mid, int high, std::vector<T>& values, std::vector<T>& scratch) {
  std::cout << "merge:  low = " << low << ", mid = " << mid << ", high = " << high << std::endl;
  int i=low, j=mid+1, k=low;









}
```

12.7 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 can 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.

12.8 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.

12.9 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.

12.10 Exercise: Complete the implementation

```
// 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;
}
// helper function to check if a position has already been used for this word
bool on_path(loc position, std::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 std::vector<std::string>& board, const std::string& word,
                     std::vector<loc>& path /* path leading to the current pos */ ) {









}
```

```
// Read in the letter grid, the words to search and print the results
int main(int argc, char* argv[]) {
  if (argc != 2) {
    std::cerr << "Usage: " << argv[0] << " grid-file\n";
    return 1;
  }
  std::ifstream istr(argv[1]);
  if (!istr) {
    std::cerr << "Couldn't open " << argv[1] << '\n';
    return 1;
  }
  std::vector<std::string> board;
  std::string word;
  std::vector<loc> path;           //  The sequence of locations...
  std::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;
    std::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
    std::cout << "\n** " << word << " ** ";
    if (found) {
      std::cout << "was found.  The path is \n";
      for(unsigned int i=0; i<path.size(); ++i)
        std::cout << "  " << word[i] << ":  (" << path[i].row << "," << path[i].col << ")\n";
    } else {
      std::cout << " was not found\n";
    }
  }
  return 0;
}
```

12.11 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.

12.12 Exercise: Analyzing our Nonlinear Word Search Algorithm

- What is the order notation for the number of operations?

Clearing Up Exponential Complexity

- Let's discuss a common mistake: $O(s^8)$ vs $O(8^s)$.
- Recall that in the non-linear word search, from any position there are a maximum of 8 choices, so any recursive call can lead to up to 8 more! (One for each direction)
- Remember the board is w wide, h high, and we are searching for a word with length s.
- For s=1 and an initial position, there's no recursion. Either we found the correct letter, or we didn't.
- For s=2, and an initial position (i, j), there are 8 calls: $(i-1, j-1)(i-1, j)(i-1, j+1)(i, j+1), (i+1, j+1), (i+1, j)(i+1, j-1), (i, j-1)$. This is $8^1 = 8$ calls.
- Now consider s=3. For each of the 8 positions from s=2, we can try 8 more positions. So that's $8 \times 8 = 8^2 = 64$ total calls.
- To determine s=i, we could repeat this approach recursively by looking at the previous step, s=$i-1$. Each time we add a step, we are multiplying by another 8, because every position from s=$i-1$ can try 8 more positions.
- In general, our solution looks like $8^{(s-1)} = 8^s * 8^{-1}$. Since 8^{-1} is just a constant, we can say $O(8^s)$.
- This isn't the whole picture though. Let's consider a few cases:
  - $w \times h = 50,000$, $s = 2$? $s = 4$? $s = 50,000$?
  - $w \times h = 4$, $s = 2$? $s = 4$? $s = 50,000$?
- How we would write a recursion to be $O(s^8)$?

```
int func(int s, int layer){
  if(layer==0){ return 1; }

  int ret = 0;
  //Make s calls
  for(int i=0; i<s; i++){
    ret += func(s,layer-1);
  }
  return ret;
}

func(1,8); => 1
func(2,8); => 256
func(3,8); => 6561
func(4,8); => 65536
```

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!