CSCI-1200 Data Structures — Fall 2025
Lecture 13 — Advanced Recursion

Review of Lecture 12

- Operators as non-member functions, as member functions, and as friend functions.
- Also... return by copy, return by reference, pass-by-reference, pass-by-pointer

Today's Lecture

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

13.1 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 writing both versions or 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. If the Big O Notation is different for recursive vs. iterative, there was also a more fundamental algorithm improvement in the refactor of the code!
- 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. But this is just the co-efficient in front of the dominant term, not a change in the Big O Analysis.

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

13.2 Binary Search

- Several lectures ago we talked about the recursive algorithm for binary search. Given a sorted vector of elements, check to see if a particular value is in the structure:

```
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);
}
```

- This algorithm only works with STL `vector`, because it allows *random access* - we can jump to the middle of the vector in O(1) time. STL `vector` allows use of indexing / the subscript operator `[ ]`.

  We cannot write an equally efficient algorithm for sorted data stored in an STL `list`, which is a *linked list* structure. To get to the middle of an STL `list` storing n items, we must call the iterator `operator++` $n/2$ times.

13.3 Exercise: Non-Recursive Binary Search

- Write a non-recursive version of binary search.

- What is the Big O Notation of the recursive and iterative versions of binary search?

13.4 Suggestions for Big "O" Notation

1. Assign variable(s) to the data size (problem size) that will have an impact on the running time / memory usage of the problem.
2. Study the code:
   - Identify the explicit `for` or `while` loops.
   - Identify implicit loops via function call recursion.
   - Look for calls to non-constant library/helper functions; for example, STL `sort` or STL `vector::erase`.
3. Determine the Big "O" Notation of each part, and the number of times each loop will execute.
4. Combine the parts:
   - Loops in series will *add*.
   - Nested loops will *multiply*.
   - Draw a tree or make a table to understand the pattern of recursion and combine the parts - *this can look very different for different problems!*
5. Simplify your answer.

13.5 Suggestions 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)!

13.6 Another Class Recursion Example: Merge Sort

- The overall plan:

  ```
  INPUT:   45.6  89.2  34.7  16.1  5.4  19.1  2.9  99.9  52.3  48.8
  ```

  Step 1: Split a vector in half,

  ```
          45.6  89.2  34.7  16.1  5.4    |    19.1  2.9  99.9  52.3  48.8
  ```

  Step 2: Recursively sort each half, and

  ```
          5.4  16.1  34.7  45.6  89.2    |    2.9  19.1  48.8  52.3  99.9
  ```

  Step 3: Merge/interleave/zip the two sorted halves into a single sorted vector.

  ```
  OUTPUT:  2.9  5.4  16.1  19.1  34.7  45.6  48.8  52.3  89.2  99.9
  ```

- 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.

- We can insert `std::cout` statements into the algorithm to watch the algorithm working.

13.7 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;
  //
  // PUSH_BACK SOME DATA INTO pts VECTOR
  //
  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;









}
```

13.8 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 (often called "Divide and Conquer")
  - Merging the results
- 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.
- Is random access necessary to efficiently sort using the merge sort algorithm? Can we efficiently implement merge sort for a linked list? Yes! Merge sort is also a good choice for linked lists. The implementation will disconnect & reconnect nodes, and doesn't need to use the scratch vector.

13.9 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.

13.10 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.

13.11 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;
}
```

13.12 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.

13.13 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!