15.1 Recursion: Introduction

An ***algorithm*** is a sequence of steps for solving a problem. For example, an algorithm for making lemonade is:

Figure 15.1.1: Algorithms are like recipes.

Make lemonade:

- Add sugar to pitcher
- Add lemon juice
- Add water
- Stir

Some problems can be solved using a recursive algorithm. A ***recursive algorithm*** is an algorithm that breaks the problem into smaller subproblems and applies the same algorithm to solve the smaller subproblems.

Figure 15.1.2: Mowing the lawn can be broken down into a recursive process.

- Mow the lawn
  - Mow the frontyard
    - Mow the left front
    - Mow the right front
  - Mow the backyard
    - Mow the left back
    - Mow the right back

The mowing algorithm consists of applying the mowing algorithm on smaller pieces of the yard and thus is a recursive algorithm.

At some point, a recursive algorithm must describe how to actually do something, known as the ***base case***. The mowing algorithm could thus be written as:

- *Mow* the lawn
  - *If lawn is less than 100 square meters*
    - Push the lawnmower left-to-right in adjacent rows
  - *Else*
    - Mow one half of the lawn
    - Mow the other half of the lawn

PARTICIPATION ACTIVITY 15.1.1: Recursion.

Which are recursive definitions/algorithms?

1) Helping N people:

   If N is 1, help that person.
   Else, help the first N/2 people, then help the second N/2 people.

   - True
   - False

2) Driving to the store:

   Go 1 mile.
   Turn left on Main Street.
   Go 1/2 mile.

   - True
   - False

3) Sorting envelopes by zipcode:

   If N is 1, done.
   Else, find the middle zipcode. Put all zipcodes less than the middle zipcode on the left, all greater ones on the right. Then sort the left, then sort the right.

   - True
   - False

15.2 Recursive functions

A function may call other functions, including calling itself. A function that calls itself is a ***recursive function***.

PARTICIPATION ACTIVITY 15.2.1: A recursive function example.

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

void CountDown(int countInt) {
   if (countInt <= 0) {
      cout << "Go!\n";
   }
   else {
      cout << countInt << endl;
      CountDown(countInt - 1);
   }
}

int main() {
   CountDown(2);
   return 0;
}
```

2

```
1
Go!
```


Animation content:

Static Figure:
Begin C++ Code:

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

void CountDown(int countInt) {
   if (countInt <= 0) {
      cout << "Go!\n";
   }
   else {
      cout << countInt << endl;
      CountDown(countInt - 1);
   }
}

int main() {
   CountDown(2);
   return 0;
}
```

End C++ Code.

The output console in the static figure contains 3 lines of output:
2
1
Go!

Step 1: The first call to CountDown( ) function comes from main. Each call to CountDown( ) effectively creates a new "copy" of the executing function.

The line of code, CountDown(2); in the main function, is highlighted, and a copy of the main function is created from the full static code. A copy of the CountDown function is created from the static code. The line of code, void CountDown(int countInt) { , is highlighted in both the static code and the copied function, and countInt: 2 is displayed near the copied CountDown function. The line of code, if (countInt <= 0) { , is highlighted in both the static code and the copied function. The line of code, else { , is highlighted. The line of code, cout << countInt << endl; , is highlighted in both the static code and copied function. The output console now contains one line of output:
2

Step 2: Then, the CountDown( ) function calls itself. CountDown(1) similarly creates a new "copy" of the executing function.

The line of code, CountDown(countInt - 1); , is highlighted in both the static code and the first copied CountDown function. The line of code, void CountDown(int countInt) { , is highlighted in the static code only, and a second copy of the CountDown function is created. countInt: 1 is displayed near the second copied CountDown function. The lines of code, void CountDown(int countInt) { ,
    if (countInt <= 0) { ,
and
    else { ,
are highlighted in both the static code and the second copied CountDown function. The line of code, cout << countInt << endl; , is highlighted in both the static code and the second copied CountDown function. The output console now contains two lines of output:
2
1

Step 3: CountDown( ) function calls itself once more.

The line of code, CountDown(countInt - 1); , is highlighted in both the static code and the second copied CountDown function. The line of code, void CountDown(int countInt) { , is highlighted in the static code only. A third copy of the CountDown function is created from the static code, and countInt: 0 is displayed near the third copied CountDown function. The lines of code void countDown(int countInt) { ,
    if (countInt <= 0) { ,
and cout << "Go!\n"; are highlighted in both the static code and the third copied CountDown function. The output console now contains 3 lines of output:
2
1
Go!

Step 4: That last instance does not call CountDown( ) again, but instead returns. As each instance returns, that copy is deleted.

The ending curly brace of the CountDown function is highlighted both in the static code and the third copied CountDown function. The third copied CountDown function is removed. The ending curly brace of the else and the ending curly brace of the CountDown function is highlighted in both the static code and second copied CountDown function. The second copied CountDown function is removed.
The ending curly brace of the else and the ending curly brace of the CountDown function is highlighted in both the static code and first copied CountDown function. The first copied CountDown function is removed. The ending curly brace of the main function is highlighted in both the static code and the copied main function. Then, the copied main function is removed.

Animation captions:

1. The first call to CountDown( ) function comes from main. Each call to CountDown( ) effectively creates a new "copy" of the executing function, as shown on the right.
2. Then, the CountDown( ) function calls itself. CountDown(1) similarly creates a new "copy" of the executing function.
3. CountDown( ) function calls itself once more.
4. That last instance does not call CountDown( ) again, but instead returns. As each instance returns, that copy is deleted.

Each call to CountDown() effectively creates a new "copy" of the executing function, as shown on the right. Returning deletes that copy.

The example is for demonstrating recursion; counting down is otherwise better implemented with a loop.

Recursion may be direct, such as f() itself calling f(), or indirect, such as f() calling g() and g() calling f().

PARTICIPATION ACTIVITY 15.2.2: Thinking about recursion.

Refer to the above CountDown example for the following.

1) How many times is CountDown() called if main() calls CountDown(5)?

Check Show answer

2) How many times is CountDown() called if main() calls CountDown(0)?

Check Show answer

3) Is there a difference in how we define the parameters of a recursive versus non-recursive function? Answer yes or no.

Check Show answer

CHALLENGE ACTIVITY 15.2.1: Calling a recursive function.

Write a statement that calls the recursive function BackwardsAlphabet() with parameter startingLetter.

Learn how our autograder works

539740.3879454.qx3zqy7

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

void BackwardsAlphabet(char currLetter){
   if (currLetter == 'a') {
      cout << currLetter << endl;
   }
   else{
      cout << currLetter << " ";
      BackwardsAlphabet(currLetter - 1);
   }
}

int main() {
   char startingLetter;
```

Run

15.3 Recursive algorithm: Search

Recursive search (general)

Consider a guessing game program where a friend thinks of a number from 0 to 100 and you try to guess the number, with the friend telling you to guess higher or lower until you guess correctly. What algorithm would you use to minimize the number of guesses?

A first try might implement an algorithm that simply guesses in increments of 1:

- Is it 0? Higher
- Is it 1? Higher
- Is it 2? Higher

This algorithm requires too many guesses (50 on average). A second try might implement an algorithm that guesses by 10s and then by 1s:

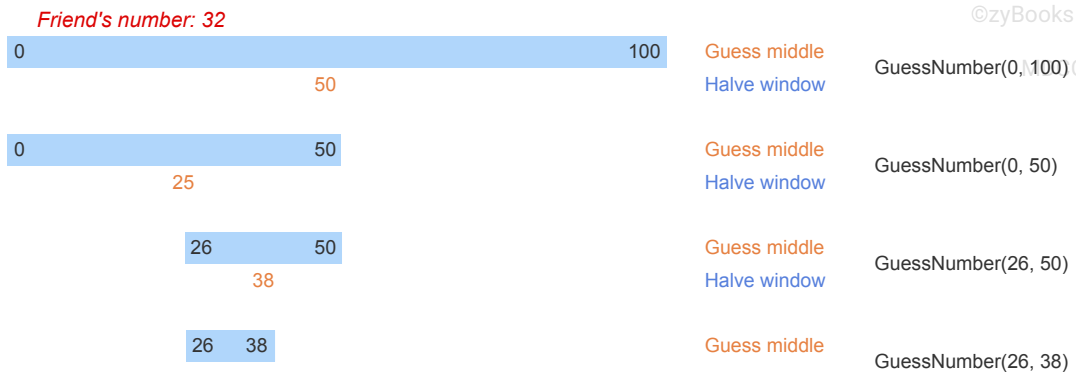
- Is it 10? Higher
- Is it 20? Higher
- Is it 30? Lower
- Is it 21? Higher
- Is it 22? Higher
- Is it 23? Higher

This algorithm does better but still requires about 10 guesses on average: 5 to find the correct tens digit and 5 to guess the correct ones digit. An even better algorithm uses a binary search. A ***binary search*** algorithm begins at the midpoint of the range and halves the range after each guess. For example:

- Is it 50 (the middle of 0-100)? Lower
- Is it 25 (the middle of 0-50)? Higher
- Is it 38 (the middle of 26-50)? Lower
- Is it 32 (the middle of 26-38)?

After each guess, the binary search algorithm is applied again, but on a smaller range, i.e., the algorithm is recursive.

PARTICIPATION ACTIVITY 15.3.1: Binary search: A well-known recursive algorithm.



32 *Correct*

Animation content:

Static Figure: Given on the left is the number 32, labeled "Friend's number". Underneath the given value are 4 separate rectangles representing different number ranges. The first range, represented by a long rectangle, shows 0 to 100, with 0 at the beginning and 100 at the end of the rectangle. The midpoint is also labeled with the number 50. On the right of this range are the binary search algorithm steps, "Guess middle" and "Halve window." To the right of this is the function call, GuessNumber(0, 100). The second range, shows 0 to 50, with 0 at the beginning and 50 at the end of the rectangle. The midpoint is also labeled with the number 25. On the right of this range are the binary search algorithm steps, "Guess middle" and "Halve window." To the right of this is the function call, GuessNumber(0, 50). The third range, shows 26 to 50, with 26 at the beginning and 50 at the end of the rectangle. The midpoint is also labeled with the number 38. On the right of this range are the binary search algorithm steps, "Guess middle" and "Halve window." To the right of this is the function call, GuessNumber(26, 50). The fourth range, shows 26 to 38, with 26 at the beginning and 38 at the end of the rectangle. The midpoint is also labeled with the number 32. On the right of this range are the binary search algorithm steps, "Guess middle" and an output "Correct." To the right of this is the function call, GuessNumber(26, 38).

Step 1: A friend thinks of a number from 0 to 100 and you try to guess the number, with the friend telling you to guess higher or lower until you guess correctly.
Step 2: Using a binary search algorithm, you begin at the midpoint of the lower range. (highVal + lowVal) / 2 = (100 + 0) / 2, or 50.
Step 3: The number is lower. The algorithm divides the range in half, then chooses the midpoint of that range.
Step 4: After each guess, the binary search algorithm is applied, halving the range and guessing the midpoint or the corresponding range.
Step 5: A recursive function is a natural match for the recursive binary search algorithm. A function GuessNumber(lowVal, highVal) has parameters that indicate the low and high sides of the guessing range.

Animation captions:

1. A friend thinks of a number from 0 to 100 and you try to guess the number, with the friend telling you to guess higher or lower until you guess correctly.
2. Using a binary search algorithm, you begin at the midpoint of the lower range. (highVal + lowVal) / 2 = (100 + 0) / 2, or 50.
3. The number is lower. The algorithm divides the range in half, then chooses the midpoint of that range.
4. After each guess, the binary search algorithm is applied, halving the range and guessing the midpoint or the corresponding range.
5. A recursive function is a natural match for the recursive binary search algorithm. A function GuessNumber(lowVal, highVal) has parameters that indicate the low and high sides of the guessing range.

©zyBooks 01/31/24 17:57 1939727 Rob Daglio MDCCOP2335Spring2024

Recursive search function

A recursive function is a natural match for the recursive binary search algorithm. A function GuessNumber(lowVal, highVal) has parameters that indicate the low and high sides of the guessing range. The function guesses at the midpoint of the

range. If the user says lower, the function calls GuessNumber(lowVal, midVal). If the user says higher, the function calls GuessNumber(midVal + 1, highVal).

The recursive function has an if-else statement. The if branch ends the recursion, known as the ***base case***. The else branch has recursive calls. Such an if-else pattern is common in recursive functions.

Figure 15.3.1: A recursive function carrying out a binary search algorithm.


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

void GuessNumber(int lowVal, int highVal) {
   int midVal;            // Midpoint of low and high value
   char userAnswer;       // User response

   midVal = (highVal + lowVal) / 2;

   // Prompt user for input
   cout << "Is it " << midVal << "? (l/h/y): ";
   cin >> userAnswer;

   if( (userAnswer != 'l') && (userAnswer != 'h') ) { // Base case: found
number
      cout << "Thank you!" << endl;
   }
   else {                                            // Recursive case:
split into lower OR upper half
      if (userAnswer == 'l') {                       // Guess in lower half
         GuessNumber(lowVal, midVal);                // Recursive call
      }
      else {                                         // Guess in upper half
         GuessNumber(midVal + 1, highVal);           // Recursive call
      }
   }
}

int main() {
   // Print game objective, user input commands
   cout << "Choose a number from 0 to 100." << endl;
   cout << "Answer with:" << endl;
   cout << "   l (your num is lower)" << endl;
   cout << "   h (your num is higher)" << endl;
   cout << "   any other key (guess is right)." << endl;

   // Call recursive function to guess number
   GuessNumber(0, 100);

   return 0;
}
```

```
Choose a number from 0 to 100.
Answer with:
   l (your num is lower)
   h (your num is higher)
   any other key (guess is right).
Is it 50? (l/h/y): l
Is it 25? (l/h/y): h
Is it 38? (l/h/y): l
Is it 32? (l/h/y): y
Thank you!
```


Calculating the middle value

Because midVal has already been checked, it need not be part of the new window, so midVal + 1 rather than midVal is used for the window's new low side, or midVal - 1 for the window's new high side. But the midVal - 1 can have the drawback of a non-intuitive base case (i.e., midVal < lowVal, because if the current window is say 4..5, midVal is 4, so the new window would be 4..4-1, or 4..3). rangeSize == 1 is likely more intuitive, and thus the algorithm uses midVal rather than midVal - 1. However, the algorithm uses midVal + 1 when searching higher, due to integer rounding. In particular, for window 99..100, midVal is 99 ((99 + 100) / 2 = 99.5, rounded to 99 due to truncation of the fraction in integer division). So the next window would again be 99..100, and the algorithm would repeat with this window forever. midVal + 1 prevents the problem, and doesn't miss any numbers because midVal was checked and thus need not be part of the window.

PARTICIPATION ACTIVITY 15.3.2: Binary search tree tool.

The following program guesses the hidden number known by the user. Assume the hidden number is 63.

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

void Find(int low, int high) {
   int mid; // Midpoint of low..high
   char answer;

   mid = (high + low) / 2;

   cout << "Is it " << mid << "? (l/h/y): ";
   cin >> answer;

   if((answer != 'l') &&
      (answer != 'h')) { // Base case:
      cout << "Thank you!" << endl;  // Found number!
   }
   else { // Recursive case: Guess in
          // lower or upper half of range
      if (answer == 'l') { // Guess in lower half
         Find(low, mid); // Recursive call
      }
      else {               // Guess in upper half
         Find(mid + 1, high);     // Recursive call
      }
   }

   return;
}

int main() {
   cout << "Choose a number from 0 to 100." << endl;
   cout << "Answer with:" << endl;
   cout << "   l (your num is lower)" << endl;
   cout << "   h (your num is higher)" << endl;
   cout << "   any other key (guess is right)." << endl;

   Find(0, 100);

   return 0;
}
```

→ main()

```
int main() {
   cout << "Choose a number from 0 to 100." << endl;
   cout << "Answer with:" << endl;
   cout << "   l (your num is lower)" << endl;
   cout << "   h (your num is higher)" << endl;
   cout << "   any other key (guess is right)." << endl;

   Find(0, 100);

   return 0;
}
```


Recursively searching a sorted list

Search is commonly performed to quickly find an item in a sorted list stored in an array or vector. Consider a list of attendees at a conference, whose names have been stored in alphabetical order in an array or vector. The following quickly determines whether a particular person is in attendance.

FindMatch() restricts its search to elements within the range lowVal to highVal. main() initially passes a range of the entire list: 0 to (list size - 1). FindMatch() compares to the middle element, returning that element's position if matching. If not matching, FindMatch() checks if the window's size is just one element, returning -1 in that case to indicate the item was not found. If neither of those two base cases are satisfied, then FindMatch() recursively searches either the lower or upper half of the range as appropriate.

Figure 15.3.2: Recursively searching a sorted list.

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

/* Finds index of string in vector of strings, else -1.
   Searches only with index range low to high
   Note: Upper/lower case characters matter
*/

int FindMatch(vector<string> stringsList, string itemMatch, int lowVal, int
highVal) {
   int midVal;         // Midpoint of low and high values
   int itemPos;        // Position where item found, -1 if not found
   int rangeSize;      // Remaining range of values to search for match

   rangeSize = (highVal - lowVal) + 1;
   midVal = (highVal + lowVal) / 2;

   if (itemMatch == stringsList.at(midVal)) {    // Base case 1: item found
at midVal position
      itemPos = midVal;
   }
   else if (rangeSize == 1) {                    // Base case 2: match not
found
      itemPos = -1;
   }
   else {                                        // Recursive case: search
lower or upper half
      if (itemMatch < stringsList.at(midVal)) { // Search lower half,
recursive call
         itemPos = FindMatch(stringsList, itemMatch, lowVal, midVal);
      }
      else {                                     // Search upper half,
recursive call
         itemPos = FindMatch(stringsList, itemMatch, midVal + 1, highVal);
      }
   }

   return itemPos;
}

int main() {
   vector<string> attendeesList(0); // List of attendees
   string attendeeName;             // Name of attendee to match
   int matchPos;                    // Matched position in attendee list

   // Omitting part of program that adds attendees
   // Instead, we insert some sample attendees in sorted order
   attendeesList.push_back("Adams, Mary");
   attendeesList.push_back("Carver, Michael");
   attendeesList.push_back("Domer, Hugo");
   attendeesList.push_back("Fredericks, Carlos");
   attendeesList.push_back("Li, Jie");

   // Prompt user to enter a name to find
   cout << "Enter person's name: Last, First: ";
   getline(cin, attendeeName); // Use getline to allow space in name

   // Call function to match name, output results
   matchPos = FindMatch(attendeesList, attendeeName, 0, attendeesList.size()
- 1);
   if (matchPos >= 0) {
      cout << "Found at position " << matchPos << "." << endl;
   }
   else {
      cout << "Not found. " << endl;
   }

   return 0;
}
```

```
Enter person's name: Last, First: Meeks, Stan
Not found.

...

Enter person's name: Last, First: Adams, Mary
Found at position 0.

...

Enter person's name: Last, First: Li, Jie
Found at position 4.
```


PARTICIPATION ACTIVITY 15.3.3: Recursive search algorithm.

Consider the above FindMatch() function for finding an item in a sorted list.

1) If a sorted list has elements 0 to 50 and the item being searched for is at element 6, how many times will FindMatch() be called?

Check Show answer

2) If an alphabetically ascending list has elements 0 to 50, and the item at element 0 is "Bananas", how many calls to FindMatch() will be made during the failed search for "Apples"?

Check Show answer

PARTICIPATION ACTIVITY 15.3.4: Recursive calls.

A list has 5 elements numbered 0 to 4, with these letter values: 0: A, 1: B, 2: D, 3: E, 4: F.

1) To search for item C, the first call is FindMatch(0, 4). What is the second call to FindMatch()?

- FindMatch(0, 0)
- FindMatch(0, 2)
- FindMatch(3, 4)

2) In searching for item C, FindMatch(0, 2) is called. What happens next?

- Base case 1: item found at midVal.
- Base case 2: rangeSize == 1, so no match.
- Recursive call: FindMatch(2, 2)

CHALLENGE ACTIVITY 15.3.1: Enter the output of binary search.

539740.3879454.qx3zqy7

Start

Type the program's output

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

void FindNumber(int number, int lowVal, int highVal) {
   int midVal;

   midVal = (highVal + lowVal) / 2;
   cout << number;
   cout << " ";
   cout << midVal;

   if (number == midVal) {
      cout << " m" << endl;
   }
   else {
      if (number < midVal) {
         cout << " n" << endl;
         FindNumber(number, lowVal, midVal);
      }
      else {
         cout << " o" << endl;
         FindNumber(number, midVal + 1, highVal);
      }
   }
}

int main() {
   int number;

   cin >> number;
   FindNumber(number, 0, 12);

   return 0;
}
```

Input

0

Output

```
0 6 n
0 3 n
0 1 n
0 0 m
```

1 2 3

Check Next

CHALLENGE ACTIVITY 15.3.2: Recursive algorithm: Search.

539740.3879454.qx3zqy7

Start

Integer numData is read from input. Then, numData alphabetically sorted strings are read from input and each string is appe to a vector. Complete the FindMatch() function, which outputs rangeSize, middleIndex, and the element at middleIndex:

- Assign rangeSize with the total number of vector elements from lowerIndex to upperIndex (both inclusive).
- Assign middleIndex with the result of dividing the sum of lowerIndex and upperIndex by 2.

▶ **Click here for example**

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

void FindMatch(vector<string> stateItems, int lowerIndex, int upperIndex) {
   int middleIndex;
   int rangeSize;

   /* Your code goes here */

   cout << "Number of elements in the range: " << rangeSize << endl;
   cout << "Middle index: " << middleIndex << endl;
   cout << "Element at middle index: " << stateItems.at(middleIndex) << endl;
}
```


1 2 3

Check Next level

Exploring further:

- Binary search from GeeksforGeeks.org

15.4 Adding output statements for debugging

Recursive functions can be particularly challenging to debug. Adding output statements can be helpful. Furthermore, an additional trick is to indent the print statements to show the current depth of recursion. The following program adds a parameter indent to a FindMatch() function that searches a sorted list for an item. All of FindMatch()'s print statements start with `cout << indentAmt << ...`. Indent is typically some number of spaces. main() sets indent to three spaces. Each recursive call *adds* three more spaces. Note how the output now clearly shows the recursion depth.

Figure 15.4.1: Output statements can help debug recursive functions, especially if indented based on recursion depth.

©zyBooks 01/31/24 17:57 1939727
Rob Daglio
MDCCOP2335Spring2024

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

/* Finds index of string in vector of strings, else -1.
   Searches only with index range low to high
   Note: Upper/lower case characters matter
*/

int FindMatch(vector<string> stringsList, string itemMatch,
              int lowVal, int highVal, string indentAmt) { // indentAmt used
for print debug

   int midVal;         // Midpoint of low and high values
   int itemPos;        // Position where item found, -1 if not found
   int rangeSize;      // Remaining range of values to search for match

   cout << indentAmt << "Find() range " << lowVal << " " << highVal << endl;

   rangeSize = (highVal - lowVal) + 1;
   midVal = (highVal + lowVal) / 2;

   if (itemMatch == stringsList.at(midVal)) {   // Base case 1: item found
at midVal position
      cout << indentAmt << "Found person." << endl;
      itemPos = midVal;
   }
   else if (rangeSize == 1) {                   // Base case 2: match not
found
      cout << indentAmt << "Person not found." << endl;
      itemPos = -1;
   }
   else {                                       // Recursive case: Search
lower or upper half
      if (itemMatch < stringsList.at(midVal)) { // Search lower half,
recursive call
         cout << indentAmt << "Searching lower half." << endl;
         itemPos = FindMatch(stringsList, itemMatch, lowVal, midVal,
indentAmt + "    ");
      }
      else {                                    // Search upper half,
recursive call
         cout << indentAmt << "Searching upper half." << endl;
         itemPos = FindMatch(stringsList, itemMatch, midVal + 1, highVal,
indentAmt + "    ");
      }
   }

   cout << indentAmt << "Returning pos = " << itemPos << "." << endl;
   return itemPos;
}

int main() {
   vector<string> attendeesList(0); // List of attendees
   string attendeeName;             // Name of attendee to match
   int matchPos;                    // Matched position in attendee list

   // Omitting part of program that adds attendees
   // Instead, we insert some sample attendees in sorted order
   attendeesList.push_back("Adams, Mary");
   attendeesList.push_back("Carver, Michael");
   attendeesList.push_back("Domer, Hugo");
   attendeesList.push_back("Fredericks, Carlos");
   attendeesList.push_back("Li, Jie");

   // Prompt user to enter a name to find
   cout << "Enter person's name: Last, First: ";
   getline(cin, attendeeName); // Use getline to allow space in name

   // Call function to match name, output results
   matchPos = FindMatch(attendeesList, attendeeName, 0
```

```
   matchPos = FindMatch(attendeesList, attendeeName, 0,
                        attendeesList.size() - 1, "    ");
   if (matchPos >= 0) {
      cout << "Found at position " << matchPos << "." << endl;
   }
   else {
      cout << "Not found. " << endl;
   }

   return 0;
}
```

```
Enter person's name: Last, First: Meeks, Stan
   Find() range 0 4
   Searching upper half.
      Find() range 3 4
      Searching upper half.
         Find() range 4 4
         Person not found.
         Returning pos = -1.
      Returning pos = -1.
   Returning pos = -1.
Not found.

...

Enter person's name: Last, First: Adams, Mary
   Find() range 0 4
   Searching lower half.
      Find() range 0 2
      Searching lower half.
         Find() range 0 1
         Found person.
         Returning pos = 0.
      Returning pos = 0.
   Returning pos = 0.
Found at position 0.
```


Some programmers like to leave the output statements in the code, commenting them out with "//" when not in use. The statements actually serve as a form of comment as well.

More advanced techniques for handling debug output exist too, such as ***conditional compilation*** (beyond this section's scope).

PARTICIPATION ACTIVITY 15.4.1: Recursive debug statements.

Refer to the above code using indented output statements.

1) The above debug approach requires an extra parameter be passed to indicate the amount of indentation.
   - True
   - False

2) Each recursive call should add a few spaces to the indent parameter.
   - True
   - False

3) The function should remove a few spaces from the indent parameter before returning.

- ○ True
- ○ False

zyDE 15.4.1: Output statements in a recursive function.

- Run the recursive program, and observe the output statements for debugging, and that the person is correctly not found.
- Introduce an error by changing `itemPos = -1` to `itemPos = 0` in the range size == 1 base case.
- Run the program, notice how the indented print statements help isolate the error of the person incorrectly being found.

Load default template... Run

```
1
2  #include <iostream>
3  #include <string>
4  #include <vector>
5
6  using namespace std;
7
8  /* Finds index of string
9     Searches only with in
10    Note: Upper/lower cas
11 */
12
13 int FindMatch(vector<str
14                int lowVal
15
```

15.5 Creating a recursive function

Creating a recursive function can be accomplished in two steps.

- **Write the base case** -- Every recursive function must have a case that returns a value without performing a recursive call. That case is called the ***base case***. A programmer may write that part of the function first, and then test. There may be multiple base cases.
- **Write the recursive case** -- The programmer then adds the recursive case to the function.

The following illustrates a simple function that computes the factorial of N (i.e. N!). The base case is N = 1 or 1! which evaluates to 1. The base case is written as `if (N <= 1) { fact = 1; }`. The recursive case is used for N > 1, and written as `else { fact = N * NFact( N - 1 ); }`.

PARTICIPATION ACTIVITY 15.5.1: Writing a recursive function for factorial: First write the base case, then add the recursive case.

```
int NFact(int N) {
   int factResult;

   if (N <= 1) { // Base case
      factResult = 1;
   }
   // FIXME: Finish
   return factResult;
}

// main(): Get N, print NFact(N)
```

```
Enter N: 1
N! is: 1
```



```
int NFact(int N) {
   int factResult;

   if (N <= 1) { // Base case
      factResult = 1;
   }
   else {        // Recursive case
      factResult = N * NFact(N - 1);
   }
   return factResult;
}

// main(): Get N, print NFact(N)
```

```
Enter N: 1
N! is: 1

Enter N: 6
N! is: 720
```

Animation content:

Static figure:
Begin Java code:
int NFact(int N) {
  int factResult;

  if (N <= 1) { // Base case
    factResult = 1;
  }
  // FIXME: Finish
  return factResult;
}

// main(): Get N, print NFact(N)
End Java code.
An output console is displayed. The text, Enter N:1, and N! is 1 is displayed within the first output console.

Begin Java code:
int NFact(int N) {
  int factResult;

  if (N <= 1) { // Base case
    factResult = 1;
  }
  else(// Recursive case
    factResult = N * Nfact(N – 1);
  }
return factResult;
}

// main(): Get N, print NFact(N)
End Java code.
A second output console is displayed. The text, Enter N:1, N! is 1, Enter N: 6, and N! is 720 is displayed within the second output console.

Step 1: The base case, which returns a value without performing a recursive call, appear and is tested first. If N is less than or equal to 1, then the nFact() method returns 1. The output console contains two lines of output:
Enter N: 1
N! is: 1
Step 2: Next the recursive case, which calls itself, appears and is tested. If N is greater than 1, then the nFact() method returns N * nFact(N - 1). The comment in the code block, // FIXME: Finish, is replaced with the lines of code, else { // Recursive case, factResult = N * nFact(N-1);, }, and the output console now contains 4 lines of output:
Enter N: 1
N! is: 1
Enter N: 6
N! is: 720

Animation captions:

1. The base case, which returns a value without performing a recursive call, is written and tested first. If N is less than or equal to 1, then the NFact() function returns 1.
2. Next the recursive case, which calls itself, is written and tested. If N is greater than 1, then the NFact() function returns N * NFact(N - 1).

A common error is to not cover all possible base cases in a recursive function. Another common error is to write a recursive function that doesn't always reach a base case. Both errors may lead to infinite recursion, causing the program to fail.

Typically, programmers will use two functions for recursion. An "outer" function is intended to be called from other parts of the program, like the function `int CalcFactorial(int inVal)`. An "inner" function is intended only to be called from that outer function, for example a function `int CalcFactorialHelper(int inVal)`. The outer function may check for a valid input value, e.g., ensuring inVal is not negative, and then calling the inner function. Commonly, the inner function has parameters that are mainly of use as part of the recursion, and need not be part of the outer function, thus keeping the outer function more intuitive.

PARTICIPATION ACTIVITY 15.5.2: Creating recursion.

1) Recursive functions can be accomplished in one step, namely repeated calls to itself.
   - True
   - False

2) A recursive function with parameter N counts up from any negative number to 0. An appropriate base case would be `N == 0`.
   - True
   - False

3) A recursive function can have two base cases, such as `N == 0` returning 0, and `N == 1` returning 1.

- True
- False

Before writing a recursive function, a programmer should determine:

1. Does the problem naturally have a recursive solution?
2. Is a recursive solution better than a non-recursive solution?

For example, computing N! (N factorial) does have a natural recursive solution, but a recursive solution is not better than a non-recursive solution. The figure below illustrates how the factorial computation can be implemented as a loop. Conversely, binary search has a natural recursive solution, and that solution may be easier to understand than a non-recursive solution.

Figure 15.5.1: Non-recursive solution to compute N!

```
for (i = inputNum; i > 1; --i)
{
   facResult = facResult * i;
}
```

PARTICIPATION ACTIVITY 15.5.3: When recursion is appropriate.

1) N factorial (N!) is commonly implemented as a recursive function due to being easier to understand and executing faster than a loop implementation.

- True
- False

zyDE 15.5.1: Output statements in a recursive function.

Implement a recursive function to determine if a number is prime.
Skeletal code is provided in the IsPrime function.

Load default template... Run

```
1
2  #include <iostream>
3  using namespace std;
4
5  // Returns false if valu
6  bool IsPrime(int testVal
7  {
8     // Base case 1: 0 and
9
10    // Base case 2: testV
11
12    // Recursive Case
13       // Check if testVa
14       // Hint: use the %
15
```


CHALLENGE ACTIVITY 15.5.1: Creating a recursive function.

539740.3879454.qx3zqy7

Start

Write ComputeSum()'s base case to output " = " and total if arg is less than or equal to 2. End with a newline.

Ex: If the input is `6`, then the output is:

```
6 + 4 + 2 = 12
```

```
1  #include <iostream>
2  using namespace std;
3
4  void ComputeSum(int arg, int total) {
5     cout << arg;
6     total = total + arg;
7
8     /* Your code goes here */
9
10    else {
11       cout << " + ";
12       ComputeSum(arg - 2, total);
13    }
14 }
15
```

1 2 3

Check Next level

15.6 Recursive math functions

Fibonacci sequence

Recursive functions can solve certain math problems, such as computing the Fibonacci sequence. The ***Fibonacci sequence*** is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, etc.; starting with 0, 1, the pattern is to compute the next number by adding the previous two numbers.

Below is a program that outputs the Fibonacci sequence values step-by-step, for a user-entered number of steps. The base case is that the program has output the requested number of steps. The recursive case is that the program needs to compute the number in the Fibonacci sequence.

Figure 15.6.1: Fibonacci sequence step-by-step.

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

/* Output the Fibonacci sequence step-by-step.
   Fibonacci sequence starts as:
   0 1 1 2 3 5 8 13 21 ... in which the first
   two numbers are 0 and 1 and each additional
   number is the sum of the previous two numbers
*/

void ComputeFibonacci(int fibNum1, int fibNum2, int
runCnt) {

   cout << fibNum1 << " + " << fibNum2 << " = "
        << fibNum1 + fibNum2 << endl;

   if (runCnt <= 1) { // Base case: Ran for user
specified
                      // number of steps, do nothing
   }
   else {             // Recursive case: compute next
value
      ComputeFibonacci(fibNum2, fibNum1 + fibNum2,
runCnt - 1);
   }
}

int main() {
   int runFor;       // User specified number of
values computed

   // Output program description
   cout << "This program outputs the" << endl
   << "Fibonacci sequence step-by-step," << endl
   << "starting after the first 0 and 1." << endl <<
endl;

   // Prompt user for number of values to compute
   cout << "How many steps would you like? ";
   cin >> runFor;

   // Output first two Fibonacci values, call
recursive function
   cout << "0" << endl << "1" << endl;
   ComputeFibonacci(0, 1, runFor);

   return 0;
}
```

```
This program outputs the
Fibonacci sequence step-
by-step,
starting after the first 0
and 1.

How many steps would you
like? 10
0
1
0 + 1 = 1
1 + 1 = 2
1 + 2 = 3
2 + 3 = 5
3 + 5 = 8
5 + 8 = 13
8 + 13 = 21
13 + 21 = 34
21 + 34 = 55
34 + 55 = 89
```


zyDE 15.6.1: Recursive Fibonacci.

Complete ComputeFibonacci() to return F_N, where F_0 is 0, F_1 is 1, F_2 is 1, F_3 is 2, F_4 is 3, and continuing: F_N is F_{N-1} + F_{N-2}. Hint: Base cases are N == 0 and N == 1.

Load default template...

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

int ComputeFibonacci(int N) {

   cout << "FIXME: Complete this function." << endl;
   cout << "Currently just returns 0." << endl;

   return 0;
}

int main() {
   int N;       // F_N, starts at 0
```


Run

Greatest common divisor (GCD)

Recursion can solve the greatest common divisor problem. The ***greatest common divisor*** (GCD) is the largest number that divides evenly into two numbers, e.g. GCD(12, 8) = 4. One GCD algorithm (described by Euclid around 300 BC) subtracts the smaller number from the larger number until both numbers are equal. Ex:

- GCD(12, 8): Subtract 8 from 12, yielding 4.
- GCD(4, 8): Subtract 4 from 8, yielding 4.
- GCD(4, 4): Numbers are equal, return 4.

The following recursively computes the GCD of two numbers. The base case is that the two numbers are equal, so that number is returned. The recursive case subtracts the smaller number from the larger number and then calls GCD with the new pair of numbers.

Figure 15.6.2: Calculate greatest common divisor of two numbers.

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

/* Determine the greatest common divisor
   of two numbers, e.g. GCD(8, 12) = 4
*/
int GCDCalculator(int inNum1, int inNum2) {
   int gcdVal;     // Holds GCD results

   if(inNum1 == inNum2) {    // Base case: Numbers are
equal
      gcdVal = inNum1;       // Return value
   }
   else {                    // Recursive case:
subtract smaller from larger
      if (inNum1 > inNum2) { // Call function with new
values
         gcdVal = GCDCalculator(inNum1 - inNum2,
inNum2);
      }
      else {
         gcdVal= GCDCalculator(inNum1, inNum2 -
inNum1);
      }
   }

   return gcdVal;
}

int main() {
   int gcdInput1;     // First input to GCD calc
   int gcdInput2;     // Second input to GCD calc
   int gcdOutput;     // Result of GCD

   // Print program function
   cout << "Program outputs the greatest \n"
        << "common divisor of two numbers." << endl;

   // Prompt user for input
   cout << "Enter first number: ";
   cin >> gcdInput1;

   cout << "Enter second number: ";
   cin >> gcdInput2;

   // Check user values are > 1, call recursive GCD
function
   if ((gcdInput1 < 1) || (gcdInput2 < 1)) {
      cout << "Note: Neither value can be below 1." <<
endl;
   }
   else {
      gcdOutput = GCDCalculator(gcdInput1, gcdInput2);
      cout << "Greatest common divisor = " <<
gcdOutput << endl;
   }

   return 0;
}
```

```
Program outputs the
greatest
common divisor of two
numbers.
Enter first number: 12
Enter second number: 8
Greatest common divisor
= 4

...

Program outputs the
greatest
common divisor of two
numbers.
Enter first number: 456
Enter second number:
784
Greatest common divisor
= 8

...

Program outputs the
greatest
common divisor of two
numbers.
Enter first number: 0
Enter second number: 10
Note: Neither value can
be below 1.
```


©zyBooks 01/31/24 17:57 1939727
Rob Daglio
MDCCOP2335Spring2024

PARTICIPATION ACTIVITY 15.6.1: Recursive GCD example.

1) How many calls are made to GCDCalculator() function for input values 12 and 8?

- 1
- 2
- 3

2) What is the base case for the GCD algorithm?

- When both inputs to the function are equal.
- When both inputs are greater than 1.
- When inNum1 > inNum2.

Exploring further:

- Fibonacci number from Wolfram.
- Greatest Common Divisor from Wolfram.

CHALLENGE ACTIVITY 15.6.1: Writing a recursive math function.

Write code to complete RaiseToPower(). Sample output if userBase is 4 and userExponent is 2 is shown below. Note: This example is for practicing recursion; a non-recursive function, or using the built-in function pow(), would be more common.

```
4^2 = 16
```

Learn how our autograder works

539740.3879454.qx3zqy7

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

int RaiseToPower(int baseVal, int exponentVal){
   int resultVal;

   if (exponentVal == 0) {
      resultVal = 1;
   }
   else {
      resultVal = baseVal * /* Your solution goes here  */;
   }

   return resultVal;
}
```


Run

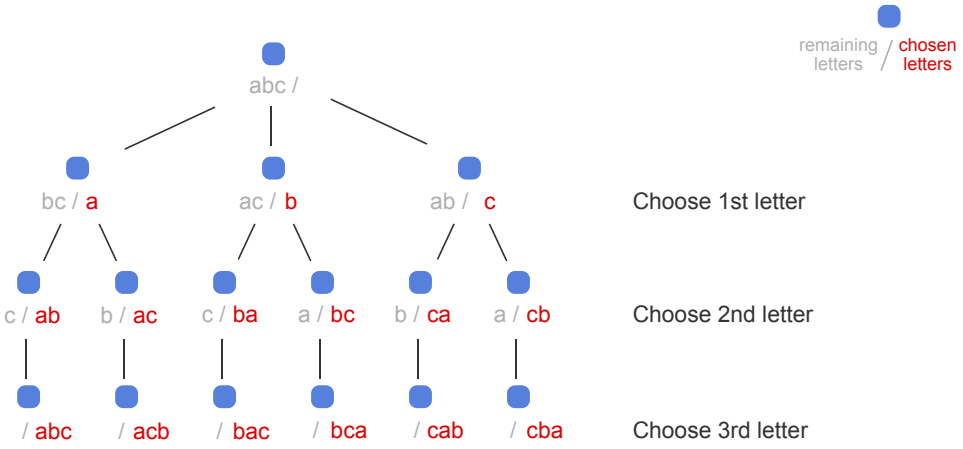
15.7 Recursive exploration of all possibilities

Recursion is a powerful technique for exploring all possibilities, such as all possible reorderings of a word's letters, all possible subsets of items, all possible paths between cities, etc. This section provides several examples.

Word scramble

Consider printing all possible combinations (or "scramblings") of a word's letters. The letters of abc can be scrambled in 6 ways: abc, acb, bac, bca, cab, cba. Those possibilities can be listed by making three choices: Choose the first letter (a, b, or c), then choose the second letter, then choose the third letter. The choices can be depicted as a tree. Each level represents a choice. Each node in the tree shows the unchosen letters on the left, and the chosen letters on the right.

PARTICIPATION ACTIVITY 15.7.1: Exploring all possibilities viewed as a tree of choices.



Animation content:

Static figure: A tree of choices with 4 levels and 16 nodes. Each node contains the remaining letters and chosen letters separated with a /. The root node contains abc/ and has 3 children. The second level containing 3 nodes has the label "Choose first letter". From left to right, the nodes contain bc/a, ac/b, and ab/c, respectively. Each node on the second level has 2 children. The third level containing 6 nodes has the label "Choose second letter". From left to right, the nodes contain c/ab, b/ac, c/ba, a/bc, b/ca, and a/cb, respectively. Each node on the third level has 1 child. The fourth level containing 6 nodes has the label "Choose third letter". From left to right, the nodes contain /abc, /acb, /bac, /bca, /cab, and /cba, respectively.

Animation captions:

1. Consider printing all possible combinations of a word's letters. Those possibilities can be listed by choosing the first letter, then the second letter, then the third letter.
2. The choices can be depicted as a tree. Each level represents a choice.
3. A recursive exploration function is a natural match to print all possible combinations of a string's letters. Each call to the function chooses from the set of unchosen letters, continuing until no unchosen letters remain.

The tree guides creation of a recursive exploration function to print all possible combinations of a string's letters. The function takes two parameters: unchosen letters, and already chosen letters. The base case is no unchosen letters, causing printing of the chosen letters. The recursive case calls the function once for each letter in the unchosen letters. The above animation depicts how the recursive algorithm traverses the tree. The tree's leaves (the bottom nodes) are the base cases.

The following program prints all possible ordering of the letters of a user-entered word.

Figure 15.7.1: Scramble a word's letters in every possible way.


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

/* Output every possible combination of a word.
   Each recursive call moves a letter from
   remainLetters to scramLetters.
*/
void ScrambleLetters(string remainLetters,  // Remaining
letters
                     string scramLetters) { // Scrambled
letters
   string tmpString; // Temp word combination
   unsigned int i;   // Loop index

   if (remainLetters.size() == 0) { // Base case: All
letters used
      cout << scramLetters << endl;
   }
   else {                           // Recursive case:
move a letter from
                                    // remaining to
scrambled letters
      for (i = 0; i < remainLetters.size(); ++i) {
         // Move letter to scrambled letters
         tmpString = remainLetters.substr(i, 1);
         remainLetters.erase(i, 1);
         scramLetters = scramLetters + tmpString;

         ScrambleLetters(remainLetters, scramLetters);

         // Put letter back in remaining letters
         remainLetters.insert(i, tmpString);
         scramLetters.erase(scramLetters.size() - 1, 1);
      }
   }
}

int main() {
   string wordScramble; // User defined word to scramble

   // Prompt user for input
   cout << "Enter a word to be scrambled: ";
   cin >> wordScramble;

   // Call recursive function
   ScrambleLetters(wordScramble, "");

   return 0;
}
```

```
Enter a word to be
scrambled: cat
cat
cta
act
atc
tca
tac
```


PARTICIPATION ACTIVITY 15.7.2: Letter scramble.

1) What is the output of ScrambleLetters("xy", "")? Determine your answer by manually tracing the code, not by running the program.

- yx xy
- xx yy xy yx
- xy yx

Shopping spree

Recursion can find all possible subsets of a set of items. Consider a shopping spree in which a person can select any 3-item subset from a larger set of items. The following program prints all possible 3-item subsets of a given larger set. The program also prints the total price of each subset.

ShoppingBagCombinations() has a parameter for the current bag contents, and a parameter for the remaining items from which to choose. The base case is that the current bag already has 3 items, which prints the items. The recursive case moves one of the remaining items to the bag, recursively calling the function, then moving the item back from the bag to the remaining items.

Figure 15.7.2: Shopping spree in which a user can fit 3 items in a shopping bag.

```
Milk  Belt
Toys  = $45
Milk  Belt
Cups  = $38
Milk  Toys
Belt  = $45
Milk  Toys
Cups  = $33
Milk  Cups
Belt  = $38
Milk  Cups
Toys  = $33
Belt  Milk
Toys  = $45
Belt  Milk
Cups  = $38
Belt  Toys
Milk  = $45
Belt  Toys
Cups  = $55
Belt  Cups
Milk  = $38
Belt  Cups
Toys  = $55
Toys  Milk
Belt  = $45
Toys  Milk
Cups  = $33
Toys  Belt
Milk  = $45
Toys  Belt
Cups  = $55
Toys  Cups
Milk  = $33
Toys  Cups
Belt  = $55
Cups  Milk
Belt  = $38
Cups  Milk
Toys  = $33
Cups  Belt
Milk  = $38
Cups  Belt
Toys  = $55
Cups  Toys
Milk  = $33
Cups  Toys
Belt  = $55
```

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

class Item {
public:
   string itemName;  // Name of item
   int priceDollars; // Price of item
};

const unsigned int MAX_ITEMS_IN_SHOPPING_BAG = 3; // Max num
items

/* Output every combination of items that fit
   in a shopping bag. Each recursive call moves
   one item into the shopping bag.
*/
void ShoppingBagCombinations(vector<Item> currBag,          //
Bag contents
                             vector<Item> remainingItems) { //
Available items
   int bagValue;        // Cost of items in shopping bag
   Item tmpGroceryItem; // Grocery item to add to bag
   unsigned int i;      // Loop index

   if (currBag.size() == MAX_ITEMS_IN_SHOPPING_BAG) { // Base
case: Shopping bag full
      bagValue = 0;
      for (i = 0; i < currBag.size(); ++i) {
         bagValue += currBag.at(i).priceDollars;
         cout << currBag.at(i).itemName << "  ";
      }
      cout << "= $" << bagValue << endl;
   }
   else {                                                 //
Recursive case: move one
      for (i = 0; i < remainingItems.size(); ++i) {     // item
to bag
         // Move item into bag
         tmpGroceryItem = remainingItems.at(i);
         remainingItems.erase(remainingItems.begin() + i);
         currBag.push_back(tmpGroceryItem);

         ShoppingBagCombinations(currBag, remainingItems);

         // Take item out of bag
         remainingItems.insert(remainingItems.begin() +
i,tmpGroceryItem);
         currBag.pop_back();
      }
   }
}

int main() {
   vector<Item> possibleItems(0); // Possible shopping items
   vector<Item> shoppingBag(0);   // Current shopping bag
   Item tmpGroceryItem;           // Temp item

   // Populate grocery with different items
   tmpGroceryItem.itemName = "Milk";
   tmpGroceryItem.priceDollars = 2;
   possibleItems.push_back(tmpGroceryItem);

   tmpGroceryItem.itemName = "Belt";
   tmpGroceryItem.priceDollars = 24;
   possibleItems.push_back(tmpGroceryItem);

   tmpGroceryItem.itemName = "Toys";
   tmpGroceryItem.priceDollars = 19;
   possibleItems.push_back(tmpGroceryItem);

   tmpGroceryItem.itemName = "Cups";
```

```
   tmpGroceryItem.itemName = "Cups";
   tmpGroceryItem.priceDollars = 12;
   possibleItems.push_back(tmpGroceryItem);

   // Try different combinations of three items
   ShoppingBagCombinations(shoppingBag, possibleItems);

   return 0;
}
```


PARTICIPATION ACTIVITY 15.7.3: All letter combinations.

1) When main() calls ShoppingBagCombinations(), how many items are in the remainingItems list?
   - None
   - 3
   - 4

2) When main() calls ShoppingBagCombinations(), how many items are in currBag list?
   - None
   - 1
   - 4

3) After main() calls ShoppingBagCombinations(), what happens first?
   - The base case prints Milk, Belt, Toys.
   - The function bags one item, makes recursive call.
   - The function bags 3 items, makes recursive call.

4) Just before ShoppingBagCombinations() returns back to main(), how many items are in the remainingItems list?
   - None
   - 4

5) How many recursive calls occur before the first combination is printed?
   - None
   - 1
   - 3

6) What happens if main() only put 2, rather than 4, items in the possibleItems list?

- Base case never executes; nothing printed.
- Infinite recursion occurs.

Traveling salesman

Recursion is useful for finding all possible paths. Suppose a salesman must travel to 3 cities: Boston, Chicago, and Los Angeles. The salesman wants to know all possible paths among those three cities, starting from any city. A recursive exploration of all travel paths can be used. The base case is that the salesman has traveled to all cities. The recursive case is to travel to a new city, explore possibilities, then return to the previous city.

Figure 15.7.3: Find distance of traveling to 3 cities.

```
Boston   Chicago   Los
Angeles   = 2971
Boston   Los Angeles
Chicago   = 4971
Chicago   Boston   Los
Angeles   = 3920
Chicago   Los Angeles
Boston   = 4971
Los Angeles   Boston
Chicago   = 3920
Los Angeles   Chicago
Boston   = 2971
```

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

const unsigned int NUM_CITIES = 3;          // Number of
cities
int cityDistances[NUM_CITIES][NUM_CITIES]; // Distance
between cities
string cityNames[NUM_CITIES];               // City
names

/* Output every possible travel path.
   Each recursive call moves to a new city.
*/
void TravelPaths(vector<int> currPath, vector<int>
needToVisit) {
   int totalDist;     // Total distance given current
path
   int tmpCity;       // Next city distance
   unsigned int i;    // Loop index

   if (currPath.size() == NUM_CITIES) { // Base case:
Visited all cities
      totalDist = 0;                    // return
total path distance
      for (i = 0; i < currPath.size(); ++i) {
         cout << cityNames[currPath.at(i)] << "   ";

         if (i > 0) {
            totalDist += cityDistances[currPath.at(i -
1)][currPath.at(i)];
         }
      }

      cout << "= " << totalDist << endl;
   }
   else {                                // Recursive
case: pick next city
      for (i = 0; i < needToVisit.size(); ++i) {
         // Add city to travel path
         tmpCity = needToVisit.at(i);
         needToVisit.erase(needToVisit.begin() + i);
         currPath.push_back(tmpCity);

         TravelPaths(currPath, needToVisit);

         // Remove city from travel path
         needToVisit.insert(needToVisit.begin() + i,
tmpCity);
         currPath.pop_back();
      }
   }
}

int main() {
   vector<int> needToVisit(0); // Cities left to visit
   vector<int> currPath(0);    // Current path traveled

   // Initialize distances array
   cityDistances[0][0] = 0;
   cityDistances[0][1] = 960;  // Boston-Chicago
   cityDistances[0][2] = 2960; // Boston-Los Angeles
   cityDistances[1][0] = 960;  // Chicago-Boston
   cityDistances[1][1] = 0;
   cityDistances[1][2] = 2011; // Chicago-Los Angeles
   cityDistances[2][0] = 2960; // Los Angeles-Boston
   cityDistances[2][1] = 2011; // Los Angeles-Chicago
   cityDistances[2][2] = 0;

   cityNames[0] = "Boston";
   cityNames[1] = "Chicago";
   cityNames[2] = "Los Angeles";
```


©zyBooks 01/31/24 17:57 1939727
Rob Daglio
MDCCOP2335Spring2024

```
   cityNames[2] = "Los Angeles";

   needToVisit.push_back(0); // Boston
   needToVisit.push_back(1); // Chicago
   needToVisit.push_back(2); // Los Angeles

   // Explore different paths
   TravelPaths(currPath, needToVisit);

   return 0;
}
```


PARTICIPATION ACTIVITY 15.7.4: Recursive exploration.

1) You wish to generate all possible 3-letter subsets from the letters in an N-letter word (N>3). Which of the above recursive functions is the closest?

- ShoppingBagCombinations
- ScrambleLetters
- main()

CHALLENGE ACTIVITY 15.7.1: Enter the output of recursive exploration.

539740.3879454.qx3zqy7

Start

Type the program's output

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

void ScrambleNums(vector<int> remainNums, vector<int> scramNums) {
   vector<int> tmpRemainNums;
   int tmpRemovedNum;
   int i;

   if (remainNums.size() == 0) {
      cout << scramNums.at(0);
      cout << scramNums.at(1);
      cout << scramNums.at(2) << endl;
   }
   else {
      for (i = 0; i < remainNums.size(); ++i) {
         tmpRemainNums = remainNums; // Make a copy.
         tmpRemovedNum = tmpRemainNums.at(i);
         tmpRemainNums.erase(tmpRemainNums.begin() + i); // Remove element at i
         scramNums.push_back(tmpRemovedNum);
         ScrambleNums(tmpRemainNums, scramNums);
         scramNums.erase(scramNums.end() - 1); // Remove last element
      }
   }
}

int main() {
   vector<int> numsToScramble;
   vector<int> resultNums;

   numsToScramble.push_back(6);
   numsToScramble.push_back(1);
   numsToScramble.push_back(8);

   ScrambleNums(numsToScramble, resultNums);

   return 0;
}
```

```
618
681
168
186
861
816
```


1 2

Check Next

CHALLENGE ACTIVITY 15.7.2: Recursive exploration of all possibilities.

539740.3879454.qx3zqy7

Start

Integer vectSize is read from input, then vectSize integers are read from input and stored in vector tokensToPick. The recursi function MakeSequence() explores all unique sequences of the tokens in tokensToPick. In MakeSequence(), write the base c to output each element in vector pickedTokens if the size of vector remainTokens is 0. Output a semicolon, ";", after each ele End with a newline.

▶ **Click here for example**

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  void MakeSequence(vector<int> remainTokens, vector<int> pickedTokens) {
6     unsigned int i;
7     int pick;
8
9     /* Your code goes here */
10
11    else {
12       for (i = 0; i < remainTokens.size(); ++i) {
13          pick = remainTokens.at(i);
```

```
14        remainTokens.erase(remainTokens.begin() + i);
15        pickedTokens.push_back(pick);
```

1 2 3

Check Next level

Exploring further:

- Recursive algorithms from khanacademy.org

15.8 Stack overflow

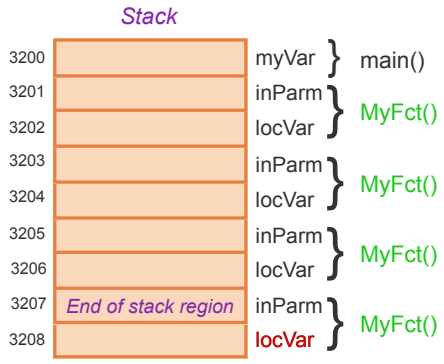
Recursion enables an elegant solution to some problems. But, for large problems, deep recursion can cause memory problems. Part of a program's memory is reserved to support function calls. Each function call places a new ***stack frame*** on the stack, for local parameters, local variables, and more function items. Upon return, the frame is deleted.

Deep recursion could fill the stack region and cause a ***stack overflow***, meaning a stack frame extends beyond the memory region allocated for stack. Stack overflow usually causes the program to crash and report an error like: segmentation fault, access violation, or bad access.

PARTICIPATION ACTIVITY 15.8.1: Recursion causing stack overflow.

```
void MyFct(int inParm) {
   int locVar;
   ...
   MyFct(...);
   ...
}

int main() {
   int myVar;
   MyFct(...);
   ...
}
```



Animation content:

Static Figure:
Begin C++ Code:
void MyFct(int inParm) {
  int locVar;
  ...
  MyFct(...);
  ...

```
}

int main() {
   int myVar;
   MyFct(...);
   ...
}
```
End C++ Code.

The static figure contains 9 memory locations, represented as adjacent rectangles, each corresponding to a memory address from 3200 to 3208. The memory locations that contain the stack region go from 3200 to 3207. Address 3208 is not part of the stack region.

Memory location 3200 is associated with the variable myVar and belongs to the stack frame of main().
Memory location 3201 is associated with variable inParm, and memory location 3202 is associated with variable locVar. Both memory locations belong to the stack frame of the first MyFct().
Memory location 3203 is associated with variable inParm, and memory location 3204 is associated with variable locVar. Both memory locations belong to the stack frame of the second MyFct().
Memory location 3205 is associated with variable inParm, and memory location 3206 is associated with variable locVar. Both memory locations belong to the stack frame of the third MyFct().
Memory location 3207 is associated with variable inParm, and memory location 3208 is associated with variable locVar, which is emphasized in a different color. Both memory locations would belong to the stack frame of the fourth MyFct(), although memory location 3208 is beyond the stack region, so the stack frame cannot be created. The memory location at 3207 is associated with the text End of stack region.

Step 1: Deep recursion may cause stack overflow, causing a program to crash.

The line of code, int main() { , is highlighted. The line of code, int myVar; in the main function is highlighted. A stack frame for main() is created, containing memory location 3200 that is associated with the variable myVar.

The line of code, MyFct(...); in the main function is highlighted. The lines of code, void MyFct(int inParm) {
   int locVar; are highlighted. A stack frame for MyFct() is created, containing memory locations 3201 and 3202 that are associated with variables inParm and locVar, respectively.

The line of code, MyFct(...); in the MyFct function is highlighted. The lines of code, void MyFct(int inParm) {
   int locVar; are highlighted. A stack frame for MyFct() is created, containing memory locations 3203 and 3204 that are associated with variables inParm and locVar, respectively.

The line of code, MyFct(...); in the MyFct function is highlighted. The lines of code, void MyFct(int inParm) {
   int locVar; are highlighted. A stack frame for MyFct() is created, containing memory locations 3205 and 3206 that are associated with variables inParm and locVar, respectively.

The line of code, MyFct(...); in the MyFct function is highlighted. The lines of code, void MyFct(int inParm) {
   int locVar; are highlighted. A stack frame for MyFct() is attempted to be created, containing memory locations 3207 and 3208 that are associated with variables inParm and locVar, respectively. locVar is highlighted in another color, since memory location 3208 is beyond the stack region, and thus the

stack frame cannot be created.

As a result, stack overflow happens and causes the program to crash.

Animation captions:

1. Deep recursion may cause stack overflow, causing a program to crash.

The animation showed a tiny stack region for easy illustration of stack overflow.

The size of parameters and local variables results in a larger stack frame. Large vectors, arrays, or strings declared as local variables, or passed by copy, can lead to faster stack overflow.

A programmer can estimate recursion depth and stack size to determine whether stack overflow might occur. Sometimes a non-recursive algorithm must be developed to avoid stack overflow.

PARTICIPATION ACTIVITY 15.8.2: Stack overflow.

1) A memory's stack region can store at most one stack frame.
   - True
   - False

2) The size of the stack is unlimited.
   - True
   - False

3) A stack overflow occurs when the stack frame for a function call extends past the end of the stack's memory.
   - True
   - False

4) The following recursive function will result in a stack overflow.

```
int RecAdder(int inValue) {
   return RecAdder(inValue + 1);
}
```

   - True
   - False

15.9 C++ example: Recursively output permutations

zyDE 15.9.1: Recursively output permutations.

The below program prints all permutations of an input string of letters, one permutation per line. Ex: The six permutations of "cab" are:

```
cab
cba
acb
abc
bca
bac
```

Below, the PermuteString function works recursively by starting with the first character and permuting the remainder of the string. The function then moves to the second character and permutes the string consisting of the first character and the third through the end of the string, and so on.

1. Run the program and input the string "cab" (without quotes) to see that the above output is produced.
2. Modify the program to print the permutations in the opposite order, and also to output a permutation count on each line.
3. Run the program again and input the string cab. Check that the output is reversed.
4. Run the program again with an input string of abcdef. Why did the program take longer to produce the results?

Load default template...

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

// FIXME: Use a static variable to count permutations

void PermuteString(string head, string tail) {
   char current;
   string newPermute;
   int len;
   int i;

   current = '?';

   len = tail.size();
```

cab

Run

zyDE 15.9.2: Recursively output permutations (solution).

Below is the solution to the above problem.

Load default template...

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

static int permutationCount = 0;

void PermuteString(string head, string tail) {
   char current;
   string newPermute;
   int len;
   int i;

   current = '?';

   len = tail.size();
```



```
cab
abcdef
```

Run

15.10 LAB: Fibonacci sequence (recursion)

The Fibonacci sequence begins with 0 and then 1 follows. All subsequent values are the sum of the previous two, for example: 0, 1, 1, 2, 3, 5, 8, 13. Complete the Fibonacci() function, which takes in an index (starting at 0), n, and returns the nth value in the sequence. Any negative index values should return -1.

Ex: If the input is:

```
7
```

the output is:

```
Fibonacci(7) is 13
```

Note: Use recursion and **DO NOT** use any loops.

539740.3879454.qx3zqy7

LAB ACTIVITY | 15.10.1: LAB: Fibonacci sequence (recursion) 0 / 10

main.cpp Load default template...

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

int Fibonacci(int n) {
```

```
5
6     /* Type your code here. */
7
8  }
9
10 int main() {
11    int startNum;
12
13    cin >> startNum;
14    cout << "Fibonacci(" << startNum << ") is " << Fibonacci(startNum) << endl
15
```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.cpp (Your program) → Output (shown below)

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

15.11 LAB: All permutations of names

Write a program that lists all ways people can line up for a photo (all permutations of a list of strings). The program will read a list of one word names into vector `nameList` (until -1), and use a recursive function to create and output all possible orderings of those names separated by a comma, one ordering per line.

When the input is:

```
Julia Lucas Mia -1
```


then the output is (must match the below ordering):

```
Julia, Lucas, Mia
Julia, Mia, Lucas
Lucas, Julia, Mia
Lucas, Mia, Julia
```

```
Mia, Julia, Lucas
Mia, Lucas, Julia
```

539740.3879454.qx3zqy7

LAB ACTIVITY | 15.11.1: LAB: All permutations of names | 0 / 10

main.cpp — Load default template...


```
#include <vector>
#include <string>
#include <iostream>

using namespace std;

// TODO: Write function to create and output all permutations of the list of
void PrintAllPermutations(const vector<string> &permList, const vector<string

}

int main() {
   vector<string> nameList;
   vector<string> permList;
   string name;
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.cpp (Your program) → Output (shown below)

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

15.12 LAB: Number pattern

Write a recursive function called PrintNumPattern() to output the following number pattern.

Given a positive integer as input (Ex: 12), subtract another positive integer (Ex: 3) continually until a negative value is reached, and then continually add the second integer until the first integer is again reached. For this lab, do not end output with a newline.

Ex. If the input is:

```
12
3
```

the output is:


```
12 9 6 3 0 -3 0 3 6 9 12
```


LAB ACTIVITY | 15.12.1: LAB: Number pattern | 0 / 10

main.cpp — Load default template...

```
#include <iostream>

using namespace std;

// TODO: Write recursive PrintNumPattern() function

int main() {
   int num1;
   int num2;

   cin >> num1;
   cin >> num2;
   PrintNumPattern(num1, num2);

   return 0;
}
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program — Input (from above) → main.cpp (Your program) → Output (shown below)

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```

15.13 LAB: Count the digits

Write a recursive function called DigitCount() that takes a non-negative integer as a parameter and returns the number of digits in the integer. Hint: The digit count increases by 1 whenever the input number is divided by 10.

Ex: If the input is:

```
345
```

the function DigitCount() returns and the program outputs:

```
3
```


LAB ACTIVITY | 15.13.1: LAB: Count the digits | 0 / 10

main.cpp — Load default template...

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

/* TODO: Write recursive DigitCount() function here. */


int main() {
   int num;
   int digits;

   cin >> num;
   digits = DigitCount(num);
   cout << digits << endl;
   return 0;
}
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **main.cpp** (Your program) → Output (shown below)

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```


15.14 LAB: Drawing a right side up triangle

Write a recursive function called DrawTriangle() that outputs lines of '*' to form a right side up isosceles triangle. Function DrawTriangle() has one parameter, an integer representing the base length of the triangle. Assume the base length is always odd and less than 20. Output 9 spaces before the first '*' on the first line for correct formatting.

Hint: The number of '*' increases by 2 for every line drawn.

Ex: If the input of the program is:

```
3
```

the function DrawTriangle() outputs:

```
          *
         ***
```

Ex: If the input of the program is:

```
19
```

the function DrawTriangle() outputs:

```
          *
         ***
        *****
       *******
      *********
     ***********
    *************
   ***************
  *****************
 *******************
*********************
```


Note: No space is output before the first '*' on the last line when the base length is 19.

LAB ACTIVITY | 15.14.1: LAB: Drawing a right side up triangle | 0 / 10

main.cpp Load default template...

```
1 #include <iostream>
```

```
using namespace std;

/* TODO: Write recursive DrawTriangle() function here. */


int main() {
   int baseLength;

   cin >> baseLength;
   DrawTriangle(baseLength);
   return 0;
}
```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.cpp (Your program) → Output (shown below)

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

15.15 LAB: Output a linked list

Write a recursive function called PrintLinkedList() that outputs the integer value of each node in a linked list. Function PrintLinkedList() has one parameter, the first node (after the head node) of a list. The main program reads the size of the linked list, followed by the values in the list. Assume the linked list has at least 1 node and that all values will be positive.

Ex: If the input of the program is:

```
5 1 2 3 4 5
```

the output of the PrintLinkedList() function is:

```
1, 2, 3, 4, 5,
```

Hint: Output the value of the current node, then call the PrintLinkedList() function repeatedly until the end of the list is reached. Refer to the IntNode class to explore any available member functions that can be used for implementing the PrintLinkedList() function.

539740.3879454.qx3zqy7

LAB ACTIVITY | 15.15.1: LAB: Output a linked list | 0 / 10

Current file: main.cpp ▾ Load default template...


```
#include <iostream>
#include "IntNode.h"
using namespace std;

/* TODO: Write recursive PrintLinkedList() function here. */


int main() {
   int size;
   int value;

   cin >> size;
   IntNode* headNode = new IntNode(-1);    // Make head node as the first node
   IntNode* lastNode = headNode;           // Node to add after
   IntNode* newNode = nullptr;                // Node to create
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.cpp (Your program) → Output (shown below)

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.