Lab 12 - Searching, Sorting, Recursion

This lab will combine multiple parts to show you:

1. How to read data from a file
2. How to search a vector
3. How to efficiently search a vector
4. How to use recursion

1. Data from a File

1. Create a new `lab12.cpp` file, and into the same directory, download the `numbers.txt` file from the course website.
   - Inspect `numbers.txt`: it contains random 100,000 random numbers between 0 and 1,000,000
2. Create a well named function to read all the integers from the `numbers.txt` file into a vector.
   - Hint: the prototype for the function might look something like:
     `vector<int> readNumbersFromFile(string fileName);`
   - Inside the function, you will want to do the following steps. See the lecture notes for example code.
     a) Open the file as an `ifstream`:
     `ifstream file(fileName);`
     b) If opening it failed, exit the program.
     Hint: *`if (file.fail()) {…}`*
     c) Read all the values from the file into a new `vector<int>`
     Hint: Use the stream extraction operator on the `ifstream` to read a single `int`
     Hint: Stop reading data (break out of the loop) when a read fails (`file.fail()`)
     d) Close the file
     e) Return the vector
3. Have `main()` call your function, and hold the returned vector in a local variable
   Hint: *`vector<int> numbers = readNumbersFromFile("numbers.txt");`*
4. Display the first 50 values in the `numbers` vector to the screen.
   - Read your screen and compare the values to the first few in the file; ensure your code is reading correctly.
5. **Understanding**
   - What is one similarity between reading from a file and reading from `cin`?
   - What are two important differences between reading from a file and reading from `cin`?

2. Finding Matches

1. In the same `.cpp` file as before, create a new function which will generate the sequence of values we want to search for:

   You function's prototype may look like:
   `vector<int> makeSequence(int size)`
   - Return a vector which is populated with the number of values in the `size` argument.
   - Make the returned sequence be all the multiples of 3. For example, the sequence of size five is {0, 3, 6, 9, 12}
2. In `main()`, generate a sequence of length 100.
3. Print out the first 50 values of your sequence to ensure it is correct.
   - *Hint: You already did this for a different vector; can you make this a function?*
4. Create a new function:
   `int countMatchesLinearly(vector<int> data1, vector<int> data2)`
   - Return the count of how many values which are in both lists.
   - *Hints:*
     a) *For each element in `data1`, see if it is in `data2`. If so, increase a count.*
     b) When vectors are unsorted, you must search through the vector one element at a time.
     c) You will likely want two nested for loops:
     outer loop: For each value, x, in data 1:
     inner loop: For each value, y, in data 2: check if x == y; if so, increase the count.
   - NOTE: If `data1` has the value 9 and `data2` happens to contain two 9's, you must **only increment the count once**.
   - Be sure to return the count.
   - This is called linear search because to find in a value `x` is in `data2`, we will look at each element of `data2` one at a time to see if it matches.
5. Test your counting function from `main()` and print the number of matches found:
   `int matches = countMatchesLinearly(sequence, numbers);`
6. Here are some results for different size sequences for comparison:
   Size 5 - found 1 matches.
   Size 10 - found 1 matches.
   Size 100 - found 6 matches.
   Size 1000 - found 93 matches.
   Size 10000 - found 948 matches
   Size 100000 - found 9496 matches.
   - You'll either need to run your program numerous times if your `main()` function only tests one size. However, you may want to make a loop to test them all (see Challenges section)!
   - Note: For large sizes (more than 10,000 elements) it may take up to a couple minutes to complete the test.

7. Calculate the time, in seconds, for how long calls to `countMatchesLinearly()` take:
   - Get the current time before calling `countMatchesLinearly()`:
     `int timeBefore = time(nullptr);`
   - After calling the `countMatchesLinearly()`, compute the elapsed time:
     `int timeTaken = time(nullptr) - timeBefore;`
8. Display how long it is taking to call `countMatchesLinearly()`. Output may be somewhat similar to the following:
   Size 5 - found 1 matches in 0s.
   Size 10 - found 1 matches in 0s.
   Size 100 - found 6 matches in 0s.
   Size 1000 - found 93 matches in 1s.
   Size 10000 - found 948 matches in 9s.
   Size 100000 - found 9496 matches in 84s.
   - Note: the times on your computer may vary a fair amount (up to a factor of 5-10 possibly) depending on processor speed and if you are in a VM or not.
9. **Understanding**
   - Explain why the first few calls are so fast, but it seems to slow down later?
   - Does the time it takes to search for the values change linearly with the size of the input sequence? (i.e., if you double the size of the input sequence, does the time double?)
   - Can you predict how many elements you'd need to search for it to take 24h?

3. Binary Search

Searching is a very common operation that happens in computer science! As seen above, the linear search is very slow (> 1 minute!) Let's do better! We'll use binary search.

Binary search is a much faster algorithm; however, it can only search data which has been sorted. So, we'll sort our data and then use binary search:

1. Create a new function named:
   `int countMatchesBinarySearch(vector<int> data1, vector<int> data2)`
2. Inside your method, first sort `data2` (which will be our unordered data from the file):
   `sort(data2.begin(), data2.end());`
   - `sort()` is defined in the algorithm header file, so you'll need:
     `#include <algorithm>`
   - `sort()` takes two parameters which are "iterators": they point to places in our vector. Here we are asking `sort()` to handle all data between the beginning and end of our vector (inclusive).

3. Since the data is now sorted, you should now use binary search to implement the rest of the function's behaviour. To search for the value `x` in vector `data2`, use:
   `bool found = binary_search(data2.begin(), data2.end(), x);`
   - *Hints*
     a) `binary_search()` is from the file include file `<algorithm>`
     b) You still need to check if each number in `data1` is in `data2`, so you'll need to loop through all of `data1`.
     c) To loop through values in `data1`, use:
        `for (int x : data1) {…}`
     d) For each value `x` in `data1`, check if `x` is in `data2` (use binary search code above)
     e) If x is in `data2`, then add 1 to the count of elements found.
4. Change you `main()` to call your new binary search matching function.
   - Rerun your program; you should see output similar to the following:
     Size 5 - found 1 matches in 0s.
     Size 10 - found 1 matches in 0s.
     Size 100 - found 6 matches in 0s.
     Size 1000 - found 93 matches in 0s.
     Size 10000 - found 948 matches in 0s.
     Size 100000 - found 9496 matches in 0s.
5. **Understanding**
   - Could we have passed `data1` by constant-reference to the function `countMatchesBinarySearch(...)`? How about `data2`? Why?
   - Analyze your results: did the binary search version find exactly the same number of matches as the linear search version? (If not, go fix your implementation!)
   - Analyze your results: describe the time needed with binary search (you do not need to understand the algorithm's implementation, just the timing results you see).
   - Can you find how big the sequence needs to be for it to take at least a couple seconds to complete all the matches using binary search? For this size, can you guess approximately how long linear search would take?
     Note: At a certain point, you'll get memory allocation errors when you make your sequence too big; try increasing its size by a factor of 10 each time you retest.
     Note: For large arrays, it takes longer to create the sequence than to search the data for the values.

4. Recursive

1. Make a new function:
   `vector<int> makeSequenceRecursive(int size) {...}`
   - Change your code to call this instead of the original iterative (loop-based) solution.
   - Base case:
     when `size` is 0, return an empty vector.
   - Recursive step:
     Return the sequence of length (`size - 1`), with an extra value appended.

   Here are hints on how to implement this function:
   - *Hint for base case:*
     Create an empty vector of `int`s, then return it.
   - *Implementation hint for base case:*
     ```
     vector<int> data;
     return data;
     ```
   - *Hint for recursive step:*
     - *Make a recursive call to `makeSequence()` for `size-1`, to give you a vector containing the start of the vector (size-1 elements). It will contain elements from the first number through to the (size-1)'th element*
     - *Compute the next value that will be in the sequence (see next hint if needed). Append this next value to your sequence and return it.*
   - *Hint for the next value that will be in the sequence: You know that the element you must add will be the `size`'th value, can you compute what it must be based on the value in `size`? Remember that the sequence is the multiples of 3. For example, for size five it is {0, 3, 6, 9, 12}.*
   - *Implementation hint for recursive step*
     ```
     vector<int> data = makeSequence(size - 1);
     int nextNumber = 2; // Change this!
     data.push_back(nextNumber);
     return data;
     ```
2. Compare the output of your program's first 50 values in the sequence generated by `makeSequenceRecursive()` and `makeSequence()`. They should match!

5. Optional Challenges

All of these challenges in this section are optional.

1. Implement your own version of binary search, as done in lecture. Do the results change? Does the timing change?
2. Create a new `makeSequenceComplex()` function to generate the following sequenced (for which we will search the input file)

   Value V_i in the sequence, where `i` is the index of the value, starting at i = 0:

   $V_i = V_{i-1} + 1$ when `i <= 5`

   $V_i = V_{i-1} + V_{(i\%5)}$ when `i > 5`

   - Use this version instead of the original and re-run the largest of the tests; does this make any difference in the timings? Explain.
3. Change to passing all vector arguments to functions by reference.
   - Use *const ref* as much as possible.
4. Make your `main()` iterate through the following sizes for the sequence:
   5, 10, 100, 1000, 10000, 100000, 1000000
   - *Hint: use a for-each loop to efficiently loop through*
     ```
     for (int size : {5, 10, 100, 1000, 10000, 100000, 1000000}) {...}
     ```
5. Change your `countMatchesLinearly()` function to be recursive:
   - *Hint idea*: Each call processes the first value in `data1` (the "head") and then calls itself recursively on the remaining elements (the "tail").
   - *Hint*: Base case is: no values `data1`.
   - *Hint*: Recursive step is:
     let `k` be 1 if the first element in `data1` is found in `data2`, otherwise 0. Return `k` + the number of times the rest of `data1` is found in `data2`.
   - Note that with recursion like this, we are in danger of overflowing the stack fro large `data1`, so start testing with a small number of elements in `data1`.
6. Create your own implementation of binary search which is implemented recursively.
   - You'll likely want use a prototype similar to:
     ```
     bool myBinarySearchRecursive(
      const vector<int> &data, int val,
      int startIdx, int endIdx);
     ```
   - When you call this function, you'll need to give it the starting index, and ending index to use. Recursive steps will narrow in these bounds.

Lab credit

Submit the following to CourSys to get credit for the lab:

- Complete the above steps. OK to skip any "optional" sections.
- In your code, type your answers to Understanding questions.
- Submit your `.cpp` file to CourSys.

