2020/11/11 - Goldbach

10 Tháng Mười Một 2020 9:03 CH

SYNOPSIS

- Go over Lab 7.
- Final Lab?

LAB 7

- As always, on Canvas, go to the "Lab 7" assignment and read the "lab7.html" file attached. All lab details will be there.

SUBMISSION COMMAND

- tar -cvf lab7.tar Goldbach.cpp

PRIME_PARTITION

- Assume the following prototype:

```
class prime_partition {
    public:
        //Constructor
        prime_partition();

        // Operator overload
        void operator()(int);

    private:
        // Private functions
        void create_pset();
```

```
    bool is_prime();
    void compute_partitions(vector<int> &, const int &, int = 0);
    void print_partitions();

    //MEMBER DATA
    set<int> pset;      //Primes 2, 3, ..., 1999
    int pset_min,       // First prime in pset
        pset_max;       // Last prime in pset
    int max_terms;      // Max primes allowed in a sum

    vector< vector<int> > partitions;
};
```

- Goal? Is simple. Given a number, find prime numbers that sum up to it.
- Upon finding a solution, don't stop. Keep going until ALL solutions are found.
  - RECALL: In Sudoku, you returned true if the recursive call returned true. Just don't do that here.
- Stop. Don't think about recursion just yet. Let's prepare.
  - main() is done for you. How sweet!

★ PREFERRED FUNCTION IMPLEMENTATION ORDER

1. is_prime
2. create_pset
3. prime_partition constructor
4. operator()
5. print_partitions
6. compute_partitions

★ DETAIL FUNCTION

- IS_PRIME
  - Ok honestly, just C+P from lab 3.
  - No cache or whatever. It just needs to return TRUE if the given number is prime, and FALSE otherwise.

- CREATE_PSET
  - Clear pset (good practice)
  - Loop from 2 to 2000 and insert numbers into pset if they are prime.
  - Set pset_min to lowest element in pset, and set pset_max to highest element.

- PRIME_PARTITION CONSTRUCTOR
  - It just calls create_pset... lol.

- OPERATOR()
  - Goldilocks CHECK the int passed in
    - AKA, if < 2 or ≥ 2000, return.
  - Set max_terms based on if given number is even or odd:
    - EVEN: 2
    - ODD: 3
  - Clear partitions.

- Make a dummy vector<int> to temporarily hold primes mid-recursion.

- Unleash recursion via compute_partitions.
  - More on this in a moment.

- Print resulting partitions via print_partitions.

- PRINT_PARTITIONS
  - So, what really is "partitions"? Scary 2D vector?
    - Yes...

    - Simply stores resulting primes that sum up to whatever number we want.

      (Ex. number = 14

      ANSWER:

      7 7

      11 3

      PARTITIONS[0] = {7, 7}
      PARTITIONS[1] = {11, 3}

  - I think you know how to print this now...

- COMPUTE_PARTITIONS

```
- void compute_partitions(
      vector<int> &numbers,
      const int   &target,
      int          sum
  ) {
      /* STUFF */
  }
```

- Conceptually:
  - We construct a sequence of prime numbers (in numbers) and, when the sum of those numbers is target, push into partitions.
  - We must abide by max_terms. So the number of primes in numbers shall not exceed this.
  - numbers[1] must be ≤ numbers[0]. Same for numbers[2] ≤ numbers[1].
    - { 13, 11, 7 } OK! { 7, 11, 13 } BAD!
    - This is to make the recursion faster and easier.
  - If a solution is found with fewer numbers than max_terms:
    - OBLITERATE partitions vector.
    - Set max_terms to numbers.size().
    - Push numbers to partitions.

- Algorithmatically:
  - If sum is equal to target...
    - If numbers.size() is under max_terms...
      - A new minimum sequence size was found.
      - Set max_terms to this new minimum
      - OBLITERATE partitions.

- Push numbers into partitions.

- return. nothing else to do.

- If sum exceeds the target or the size of numbers exceeds max_terms, return.

- Now for Sudoku-like recursion...

- Have three iterators:
  1. start - beginning of pset
  2. end - If numbers is empty, upper_bound of target. Otherwise, upper_bound of last number in numbers.
  3. ii - Use to loop from start to end.

- To "clarify", this is the point when we start putting primes from pset into numbers. But we need numbers[1] ≤ numbers[0] and so on. Thus, above, we are setting iterators to guarantee that. Hence:
  - If numbers is empty, we are unrestricted and can go from first prime (2) to whatever target is.

    (Ex. target = 14

numbers = {2}
3
5
7 ← All valid first elements.
11
13

- If numbers isn't empty, we are restricted to going up to the last number inserted into numbers.

(Ex. target = 14 (again...)

numbers = { 11, 2 }

3

5 ← All valid second values. Notice 13 is gone.

7

11

Let's assume 11 for example.

- Anyways, loop ii from start to end and:
  1. Push *ii to numbers
  2. Call compute_partitions. Here, use that optional third argument. I'm thinking about sum + *ii?
  3. Pop that number off of numbers.

- That's it!

- Confused? There's more than one way to achieve the recursion. This is my custom method which, I believe, is easier to understand, coming from Sudoku. Dr. Gregor utilised more class member variables, so his recursive function doesn't pass a vector around. Don't be afraid to experiment.