Problem

The problem is described in the attached PDF file.

Let's try to restate it and convert it to an equivalent problem.

Terminology:

- N: 0 < homes ≤ 10,000 (maximum number of homes on the block)
- S: 0 ≤ max ≤ 1000 (maximum number of pieces of candy that the child may collect)
- A: 0 ≤ pieces ≤ 1000 (representing the number of pieces given at each home)

Restatement

Given an array "A" of "N" integers and a limit "S", find a contiguous sequence
(A[left .. right] and no gaps) such that:

- R = sum(A[left .. right]) <= S
- R is maximized.
- If there are multiple such sequences (same value of R), select the first one.

Return:

- Sequence left:right
- The sum.
- If no sequence found, then return and empty result.

Beatufiy the output so that:

- If no sequence is found, print the string: "Don’t go here".
- If a sequence is found print: "Start at home {left} and go to home {right} getting {sum} pieces of candy"

Solution

We'll use a sliding window (two indexes/pointer) as following:

- Keep [left, right] and the current sum
- Move right (add elements to window)
- If the sum of the windows exceeds the max sum, then move to the left until the window's sum is less or equal to the looked up sum.
- Track the best sum and its indices.

```
struct Result {
  size_t begin{};
  size_t end{};
  uint32_t sum{std::numeric_limits<uint32_t>::max()};

  // Check if the result is valid
  bool valid() const { return sum != std::numeric_limits<uint32_t>::max(); }

  // Used for tests
  bool operator==(const Result& other) const {
    return begin == other.begin && end == other.end && sum == other.sum;
  }
};

// Finds the maximum contiguous subsequence in `input`
// such that the sum of its elements does not exceed `max_sum`.
// Returns an invalid Result if no such subsequence exists.
// Notes:
// - The subsequence is defined by the range [begin, end] (end is inclusive).
// - If multiple subsequences have the same maximum sum, the first one is
// returned.
// - The result is zero-based indexed.
static inline Result max_configuous_sequence(
    const std::span<const uint16_t> input, const uint32_t max_sum) {
  if (input.empty() || max_sum == 0) {
    return {};
  }

  uint32_t crt_sum{};
  uint32_t best_sum{};

  size_t leftIt{}, bestLeftIt{}, bestRightIt{};

  for (size_t right = 0; right < input.size(); ++right) {
    crt_sum += input[right];
```

```
    // If the sum exceeds max_sum, we need to shrink the window
    while (crt_sum > max_sum && leftIt <= right) {
      crt_sum -= input[leftIt];
      ++leftIt;
    }

    // Check if we have better results now
    if (crt_sum <= max_sum && crt_sum > best_sum) {
      best_sum = crt_sum;
      bestLeftIt = leftIt;
      bestRightIt = right;
    }
  }

  if (bestRightIt < bestLeftIt || best_sum != max_sum) {
    // No solution
    return {};
  }

  return Result{bestLeftIt, bestRightIt, best_sum};
}
```

Testing

I have written & tested the solution using c++20 (clang 18).
Attached in this folder you'll also find test.cpp that includes basic correctness tests.

To run the automated tests:

```
make tests
```

I have also included some sample files in the sample folder.

To build the main program:

```
make build_main --trace
```

To run the binary against a simple input you can do

```
SAMPLE=samples/sample1.txt make main
Start at home 2 and go to home 5 to collect 10 pieces of candy.
```

Test max speed on single CPU

Note that by default the Makefile is building the code in debug mode (`-O0`)
To build in optimized mode do

```
export CXX_FLAGS_EXTRA="-O3"
export WORKSPACE_BUILD_ROOT=/tmp
# Build
make clean && make build_main

# Test
/tmp/max_sequence_sum/main.x samples/sample1.txt
```

Subsidiary question

Parallelize the program with OpenMP for multi-core execution.

Because each iteration depends on the previous one, we can't simply use `#pragma omp parallel for`.

We would need to manually partition the input. However, there would be quite a few complications:

- The solution might be split accros 2 ore more partitions. In this case all threads will do busy work and fail.
- Another problem would be that the first solution could be accross the first few partitions but another thread also succeeds. In this case the returned solution would not be the first one found.

However, based on the problem statement, there's another clue: the result size is an order of magnitude smaller than the input size (max 10k homes, max 1k pieces given at a single home and max 1k held by a child).

Here we could imagine and approach like a tree reduction, where the tree will be composed as following:

- At the leaf level, we partition the input span with the size of max result and deal with every partition in an OMP thread (e.g `max_configuous_sequence(span_of_thread_i, max)`).
- If no solution is found by any of the threads, we double the partition size and resume the work in each thread.

Some pseudo code:

```
const size_t num_threads =  omp_get_max_threads();
size_t partition_size = input.size() / num_threads;
std::vector<Result> partitions(num_threads); // Store the results for each threa
size_t tree_level{};

// Fill out the initial partitions here per thread
// ...

// Then, while result not found or can't iterate any longer
while(partition_size <= input.size()) {
    #pragma omp parallel
    {
        auto &partition = partitions[omp_get_thread_num()];
        // call max_configuous_sequence for this partition.
        // The body of `max_configuous_sequence`  should be updated to return the last partiton that is less
        // than the max_sum as it'll be used in the next iteration
    }

    // If a result is found, return it
    for(auto &r:partitions) {
        if (r.valid()) {
            return r;
        }
```

```
    }

    // Update the partitions to include the next unincluded partition (thread = 1 + tree_level)
    // ...

    ++tree_level;
}

// Deal with the situation where the input size is not a multiple of threads size (there will be a leftover partition) ...
```

The speedup in thix case would be:

- Lucky case (e.g the solution is in the last partition): ~X = number of OMP threads (usually number of CPU cores)
- Unlucky case (e.g the solution is spread accross multiple partitions): <1 (as if done by a single thread, but with OMP overhead)