Calendars Part 2

COSC122 Assignment 2012

September 25, 2012

1 Overview

Calendars are an important part of modern life. These days, it isn't uncommon to have several portable devices with synchronised calendars. This assignment requires you to implement algorithms to search for conflicting events and merging lists of events efficiently, such as might be found in real-world calendaring systems.

This assignment is designed to help you understand both the theoretical and practical concerns related to implementing a variety of algorithms. As a computer scientist, you will often encounter problems for which no pre-existing solution exists. Your knowledge of and experience with algorithms will directly affect the quality and efficiency of systems that you produce, especially when processing problems that involve a large data-sets.

1.1 Due Dates

Part One of this assignment has two tasks which you must complete, each part is worth 20% of the total assignment. The **due date** for part 1 is **5pm, Monday the 17th of September**. This part of the assignment is not included in this document.

Part Two of this assignment has four tasks which you must complete, each part is worth 15% of the total assignment. The **due date** for part 2 is **5pm, Monday the 8th of October**.

The drop dead date is 5pm one week after each due date. Late submissions will be accepted between the due date and the drop dead date, but there will be a penalty of 15% of the maximum possible grade for late submission. No assignments will be accepted after the drop dead date.

1.2 Submission

One week before the due date, an assignment submission page will be made available. You must upload the required files as specified on that page.

We recommend that you test your code thoroughly using the supplied unit tests before making a submission. Additional unit test may be provided as required if there are any ambiguities with the assignment.

1.3 Implementation

You must write full algorithms by your own hand. Do not import any additional standard libraries unless explicitly given permission within the task outline.

1.4 Getting Help

The work in this assignment is to be carried out individually. You must not discuss code-level details with anyone other than the course tutors and lecturers. You are, however, permitted to discuss high-level details of the program with other students.

If you get stuck on a programming issue, you are encouraged to ask your tutor or the lecturer for help. You may not copy material from books, the internet, or other students. We will be checking carefully for copied work.

If you have a general assignment question or need clarification on how something should work, please use the class forum available at http://learn.canterbury.ac.nz/mod/forum/view.php?f=375.

2 Part 2

It is very typical to present multiple lists of events at the same time in a calendar. However, often these lists of events come from different sources, e.g. online calendars, offline calendars, other people's calendars, etc. This process requires that events are merged together sequentially to be presented to the user via text or a user interface of some sort.

The following tasks each use different algorithms to merge together lists of events sequentially. Task 1 requires that you implement linear 2-way merge. Task 2 investigates whether this can be improved by using a linear n-way merge. Task 3 takes the same approach but uses a n-heap to select the minimum item at each step. Task 4 reverts back to a 2-way merge but changes the order of merging to reduce the number of comparisons.

2.1 Task 1 (15%)

The file `merge_linear_2way.py` contains an incomplete implementation of a simple linear merge algorithm. To compute the merged calendar c_m from n calendars c_0, c_1, c_2, ..., c_n, we compute initial $c_m = c_0$ followed by for $i = 1$ to n: $c_m = \texttt{merge}(c_m, c_i)$. Items are merged in order of the `start_timestamp` and the first argument takes priority over the second if they both have the same `start_timestamp` (i.e. the merge must be stable). See figure 1 for a visual representation of this process.

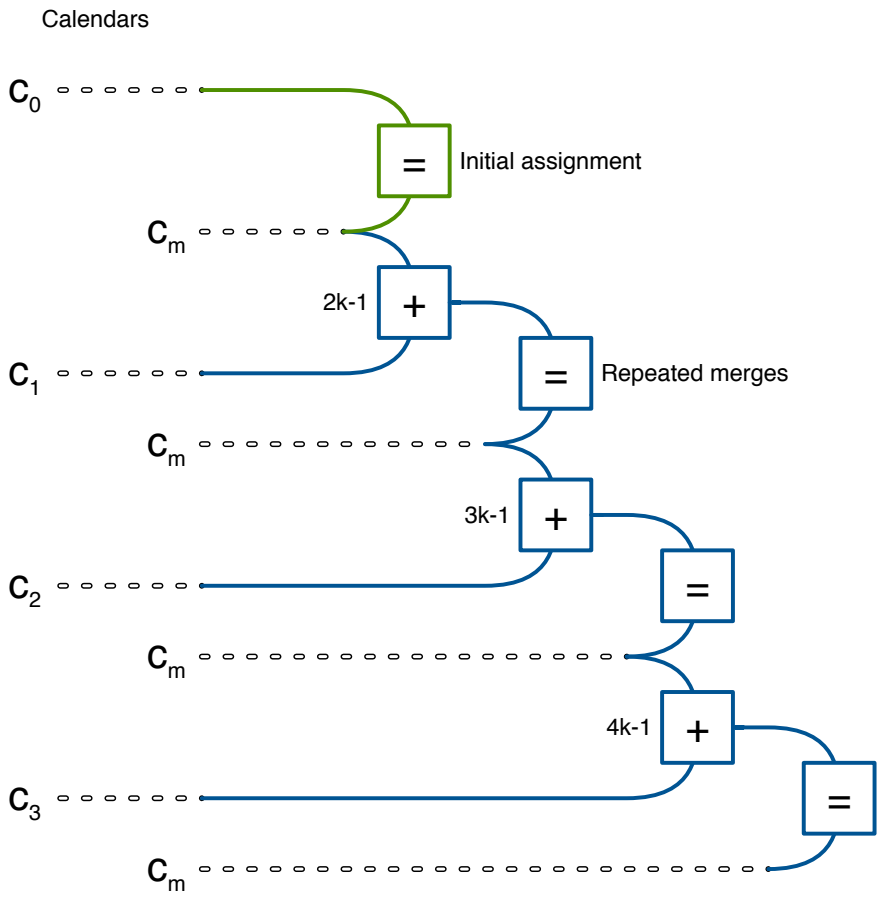

Figure 1: A visualisation of the linear 2-way merge process with 4 lists.

You must complete this implementation by adding code between the commented segments:

```
# --[ WRITE CODE HERE ]-->
# Specific details on what you need to implement will be provided.
# <--[ WRITE CODE HERE ]--
```

You can run the script from the command line like so:

```
python ./merge_linear_2way.py data/events_*.txt
```

This will print out a list of merged items in order. You can assume that the input calendars are in sorted order and within a specific calendar of events there are no overlaps.

2.1.1 Events

Calendars are lists of events. A Python module `eventlib` is provided that can load events from JSON encoded files. You should familiarise yourself with the code in `eventlib`. You should not reimplement the functions provided by `eventlib`, or change the command line interfaces provided.

The implementation must merge calendars in order such that the result is also in order based on the `start_timestamp`. If you have two events you can compare them using the standard operators <, <=, >=, > which compares two events based on the `start_timestamp`, e.g. `event1 <= event2` is equivalent to `event1.start_timestamp <= event2.start_timestamp`, but in testing the first form will be used.

2.2 Linear Merge

The linear merge algorithm steps through two calendars at a time using an index for each calendar. For `merge(left, right)` we keep two indices l and r and compare the indexed events in both calendars `left[l] < right[r]`. Based on the result of this we add either one of those events to the list `merged` and increment the appropriate counter.

The result of the computation is a combined list of events in sorted order. These, by default, are printed to `stdout`, along with timing and number of comparisons to `stderr`.

2.2.1 Comparisons

As part of your implementations, you will be asked to record the number of comparisons performed. A Python module `comparelib` is provided with a `class Counter` that is used as a global to track the number of comparisons.

A general rule of thumb is that you must call `counter.increment()` every time you compare two events. Specific instructions will be given as required in the comments.

2.2.2 Testing

A unit test `test_merge_linear_2way_01.py` is provided which can be used to test the basic functions in `merge_linear_2way.py`. You can run this file in your preferred IDE or from the command line like so:

```
python ./test_merge_linear_2way.py
```

The provided unit test covers basic functionality but it doesn't cover all edge cases. You may want to add additional checks to the unit test to verify the correct functioning of your program.

Initially, if you run this, you will get lots of errors. This is normal, as you haven't completed any parts of the implementation. As you complete various parts of the implementation, an increasing number of unit tests should pass successfully.

2.3 Task 2 (15%)

The file `merge_linear_nway.py` contains an incomplete implementation of a n-way merge algorithm. This is similar to the 2-way merge but rather than processing 2 calendars at a time, all calendars are processed in one go. To compute the merged calendar c_m from n calendars c_0, c_1, c_2, ..., c_n, we compute $c_m = \texttt{merge}([c0, c1, c2, ..., c_n])$. The merge algorithm assigns an index for each calendar and uses a linear scan over all indexed events to find the nearest event; the index for that calendar is incremented and the event added to `merged`. For n calendars with k items per calendar, this algorithm is $O(n^2k)$

You must complete this implementation similarly to Task 1. You can run this script similarly to that in Task 1. The meaning of the input files and output is identical to Task 1. The test file is called `test_merge_linear_nway_01.py`.

2.3.1 Find Minimum

The implementation requires that you implement a linear scan to find the minimum value. This is a simple loop that initially starts with `minimum_index = 0`, then for every `item[1:]` checks if the item is less than the currently found minimum, and if so updates `minimum`. In this task you will probably want to keep track of the index that the minimum value was found at.

2.4 Task 3 (15%)

The file `merge_heap_nway.py` tries to improve upon the linear scan used above by using an n-heap. For each item in task 2, we must scan n calendars to find the minimum event. However, we can use a heap to reduce this to $O(\log n)$ which makes the overal efficiency $O(nk \log n)$. Does this improve performance in practice?

You must complete this implementation similarly to Task 1. You can run this script similarly to that in Task 1. The meaning of the input files and output is identical to Task 1. The test file is called `test_merge_heap_nway_01.py`.

2.4.1 Heaps

This implementation requires the implementation of two heap-related functions `siftdown` and `heapify`. For n-heap implementations you will construct a heap of `[event, index, calendar]` lists. When you take the top item from the heap, you then know which event and calendar it came from, therefore finding the next event, if any, is easy.

Typical heap implementations might use `push` and `pop` operations, however in our case we can combine these functions together since when we remove an event we need to push the event back on the top anyway. In the case that a calendar still has items, we put the next event on the calendar, otherwise we follow the process of typical heap `pop` and use the last item in the heap.

The heaps in this implementation are indexed from 0. To calculate the first child for a given parent index use `childpos = 2*pos + 1` and to calculate the parent of a given index use `parentpos = (pos - 1) >> 1`.

2.5 Task 4 (15%)

Another approach that has efficiency $O(nk \log n)$ is the balanced binary merge. Going back to the original 2-way merge, what we notice from the diagram is that it is unbalanced. We merge from left to right, 2 calendars at a time.

Merging two lists each with k items requries at most $2k - 1$ comparisons. If our merge is unbalanced we get the worst case possible. We can improve this by merging every pair of lists, then every pair of those results, and so on until only one list remains. Figures 1 and 2 have the number of comparisons required at each merge operation. You can see for yourself that binary merge requires less comparisons for the same input.

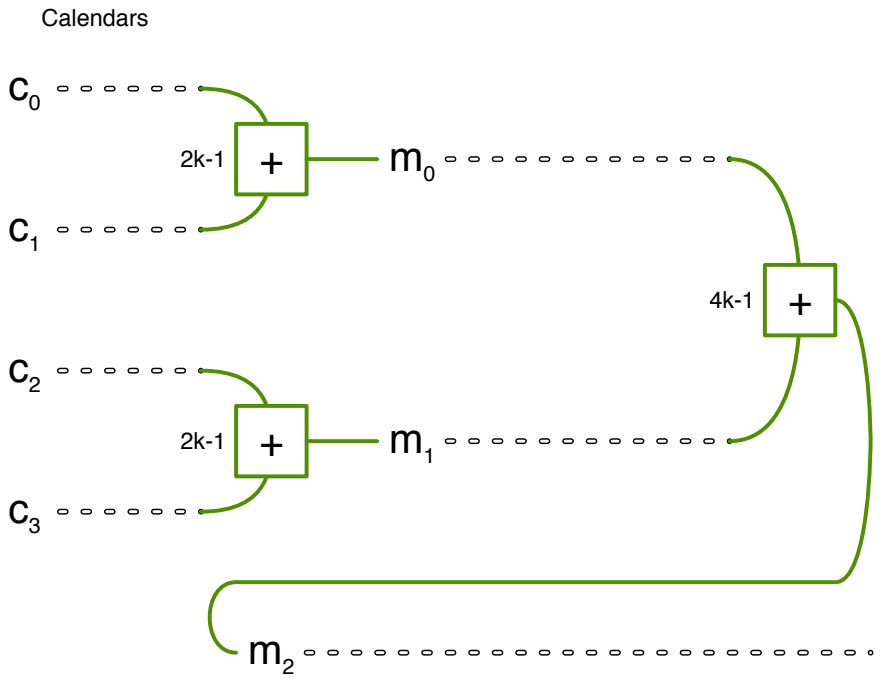

Figure 2: A visualisation of the linear 2-way merge process with 4 lists.

The file `merge_binary_2way.py` contains an incomplete implementation of a binary 2-way merge. The actual implementation is almost identical to the linear 2-way merge, but the way lists are merged must be done such that for a given list of calendars, we merge them in pairs and return the result. Once only one result remains we are finished.

You must complete this implementation similarly to Task 1. You can run this script similarly to that in Task 1. The meaning of the input files and output is identical to Task 1. The test file is called `test_merge_binary_2way_01.py`.