Informatics 2 – Introduction to Algorithms and Data Structures

Coursework 1: Simple TimSort and space-saving red-black trees

John Longley

October 22, 2022

This is a Python programming practical in two parts. In Part A, you will implement a simplified version of the algorithm known as *TimSort* (the built-in sorting algorithm for lists in Python). This borrows ideas from InsertSort and MergeSort, but applies them in a novel way. In Part B, you will implement `lookup` and `insert` operations for an implementation of dictionaries via *red-black trees* — one that will be more space-efficient than most classic tree implementations. Both parts (especially B) will require a good grasp of the organization of program memory as explained in Lecture 6.

This is a credit-bearing coursework assignment worth 15% of the overall course mark. You should submit your completed files via the Learn Assessment page: the deadline is **12 noon on Friday 11 November 2002**. The coursework will be marked out of 100, using a combination of automarker tests and eye inspection by a human marker. Your marks and feedback will be returned to you by Friday 2 December.

To get started, you will need the template files `timsort.py` and `red_black.py`, available from the Learn Assessment page. In each of these files, some Python code is provided for you, and your task is to complete the implementation by adding other function or method definitions at the points marked `# TODO`.

Some general points:

- The total volume of code you're expected to write is about **80 lines for each part**. Don't worry if your version is a little longer than this — but if you find yourself writing *much* more, you should pause to ask whether there's a simpler approach.
- Because the coursework will be largely automarked, it's very important that you comply with the instructions precisely, e.g. as regards the names of functions, what arguments they take, and what they return. It's also important that you make no changes to the provided code, and add new code only at the points indicated.
- You should add a modest number of **comments** to your code to help make it readable. These should be short and not too numerous — the skeleton code provided gives an idea of the level of commenting expected. Put yourself in the shoes of a human marker with a tight time budget of 15 minutes per student submission: you're wanting to steer their thoughts in the right direction as quickly as possible.
- It's your responsibility to **test** your code thoroughly before you submit it. You should expect to spend some time devising and constructing your own tests to ensure that

your functions work correctly in all cases. You may work mostly on your own machine, but before submitting you should do a final check that your code runs correctly on the `python3.8` installation on the DICE machines, as this is where it will be automarked.

- Please don't be surprised or disheartened if it takes some time to get your code to work! This is *perfectly normal*, even for seasoned programmers — so allow yourself a generous amount of time for debugging. Tracking down an elusive bug can be time-consuming and deeply frustrating, but this is an important practical skill that can be acquired only by experience.

- If you feel you'd benefit from help on the Python programming side, **lab sessions** will be running at the advertised times, and the demonstrators there may be able to help you. You're also welcome to post questions to the course's **Piazza forum** (selecting the 'cw1' folder), but please check through previous postings first to see whether your question has already been asked. We'll try where possible to answer queries within 48 hours. You are also encouraged to help one another (and us) by responding to queries if you know the answer, provided you observe the following...

- **STERN WARNING!!!** It is *absolutely forbidden* to share any part of a solution attempt with another student — even a one-line snippet of code — on Piazza or by any other means. This applies even if you know that your attempt is incorrect. Any breach of this rule will be treated as a very serious offence, so please err on the side of caution in the way you phrase your questions. If copying and pasting a Python error message, check that it doesn't have a fragment of your code embedded within it.

  Likewise, if you use GitHub or any similar repository service while working on the coursework, please make absolutely certain that your files are not visible to anyone else. Again, any breach of this may be treated as a case of academic misconduct.

Part A: A simple version of TimSort [50 marks]

TimSort is a general-purpose sorting algorithm for lists due to Python developer Tim Peters. It is a highly competitive approach to sorting, combining the strengths of MergeSort and InsertSort, and is the standard sorting algorithm built into Python.

As we've seen in lectures, MergeSort does much better than InsertSort in the average and worst cases ($\Theta(n \lg n)$ vs. $\Theta(n^2)$ time), but InsertSort is faster in the best case ($\Theta(n)$ time), and in general may do better if the input listed is already almost sorted. A key observation is that in real-world sorting scenarios, there are very often long runs of items that are already either sorted or reverse-sorted, and the idea of TimSort is that it's advantageous to detect and preserve these portions so that we don't have to sort them again.

The full TimSort algorithm is quite complex, involving many bells and whistles and admitting numerous variants. Here we'll content ourselves with a fairly simple version that illustrates the main idea. Our algorithm works broadly as follows:

1. In an initial phase (which is already implemented for you), the input list is split into *segments* which are of three kinds: *increasing* (i.e. already sorted), *decreasing* (i.e. reverse-sorted) and *unsorted.* These are identified by scanning through the list and looking for longish runs of increasing or decreasing elements.

2. We then process each segment separately. For increasing segments, there is nothing to do, and for decreasing ones, all we need to do is to *reverse* the segment (a linear-time operation). Each unsorted segment is sorted using an in-place InsertSort (recall that this is faster than MergeSort for shortish lists).

3. Finally, we apply a variant of MergeSort to repeatedly merge these sorted segments to form longer ones, until we're down to just a single segment. However, in contrast to traditional MergeSort which does 'even splits' as far as possible, here the segments we're merging might be of very different lengths.

There are a couple of balancing acts involved here. Ideally, we don't want our segments to be too short: a lot of fiddly merging of very short segments would be wasteful on time, and it's more efficient to group them into medium-sized segments and then InsertSort these. (An occasional short segment between two long increasing or decreasing ones may be unavoidable.) At the same time, we don't want our unsorted segments to be too long, or we lose the advantage of InsertSort. (For the increasing and decreasing segments, the longer the better.)

Open the file `timsort.py` and look at the code provided. We'll be working with an input list `L`, and the class `Segment` is used for identifying segments of `L`. Note in particular that we follow the Python convention for list slicing: a segment `s` refers to the portion of `L` from `L[s.start]` to `L[s.end-1]` inclusive, whose length is `s.len()`.

There follows some code for splitting a list `L` into segments as described above, subject to a parameter `runThreshold` specifying the minimum length for ascending or descending runs we wish to preserve, and parameters `blockMin`, `blockMax` specifying the desired size range for unsorted segments. You don't need to understand all this code in full detail — but to get a feel for what it does, temporarily set

```
runThreshold, blockMin, blockMax = 3, 4, 7
```

then try something like

```
segments([3,4,5,6,7,5,6,7,8,5,4,6,4,3,2,1,2,3,2,1,2,3,2,1,2,3])
```

This produces the output:

```
[Segment(0,5,1), Segment(5,8,1), Segment(8,11,0), Segment(11,15,-1),
Segment(15,19,0), Segment(19,23,0), Segment(23,26,1)]
```

As you can see, this has divided the whole of `L` into non-overlapping segments. After two decent ascending runs, the portion (8,11) has been classified as an unsorted segment, and this is followed by a descending run. The portion from index 15 to 22 (inclusive) is classified as unsorted, and has been split into two blocks of length 4. Feel free to experiment with other lists and other parameter values to get the idea.

You are now ready to attempt the following programming tasks.

Task 1 (15 marks). First, we wish to process the individual segments. At the point marked `# TODO: Task 1`, add code to define the following functions:

- A function `insertSort(L,start,end,key=lambda x:x)` that uses *in-place InsertSort* to sort the portion of `L` from `L[start]` to `L[end-1]` inclusive. Your code should be modelled on the pseudocode for in-place InsertSort from Lecture 2. It's not permitted to call any built-in sorting function here!

  The optional argument `key` is a function that specifies what we're sorting by. E.g. if `L` is a list of lists, and we wish to sort them by length, we can supply the function `len` as this argument. If no `key` argument is specified, the identity function is used by default, which means in effect that the list items themselves are being directly compared using `<`. This is what we want for sorting a list of integers or strings in the usual way.[1]

Your `insertSort` function should have the effect of sorting the required portion of `L`, but need not return a result.

- A function `reverse(L,start,end)` that *reverses* the portion of `L` from `L[start]` to `L[end-1]` inclusive. Again, this should work efficiently in an in-place way, avoiding the need for a second array as 'scratch space'. Don't attempt to call any built-in list reversal operation here: you must implement your own reversal.
- A function `processSegments(L,segs,key=lambda x:x)` that takes a list `L` and a list `segs` of `Segment` objects (of the kind returned by `segments`), and applies the appropriate processing to each segment in turn, as outlined above. The `key` argument plays the same role as before. After calling this function on `L` and `segments(L)`, the whole of `L` should consist of individually sorted segments, so that the tags `1,0,-1` are no longer relevant.

Task 2 (15 marks). We now develop some code for *merging* two already sorted segments of `L`, writing the result into a second list `M`, and also a simple function for *copying* a single segment from `L` to `M`.

Specifically, at the point marked `# TODO: Task 2`, you should supply:

- A function `mergeSegments(L,seg1,seg2,M,start,key=lambda x:x)` that takes a list `L` and two `Segment` objects, and writes the merged (sorted) segment into `M` beginning at position `start` (i.e. into the cells `M[start],M[start+1],...`). You may assume that `M` extends far enough to accommodate this.

  The `key` argument plays the same role as before. In case of a tie between two items with the same key, you should give precedence to the one from `seg1`, so that the resulting ordering of such items within `M` is the same as that within `L` (we may assume `seg1` appears to the left of `seg2` in `L`). This will help to ensure that our version of TimSort is *stable*: that is, it preserves the relative ordering of items with the same key.

  Your function should have the effect of writing entries into `M`, but what it *returns* should be simply the total length of the resulting merged segment. Don't attempt to return `M` itself or any portion of it.
- A function `copySegment(L,seg,M,start)` that copies a given segment of `L` into `M`, beginning at position `start`. Again, you may assume `M` has enough space to accommodate this. Your function should return the length of the copied segment.

Task 3 (15 marks). The final part (harder) is to use the above functions to repeatedly merge segments into larger ones until we have just a single sorted segment. We do this in a series of *rounds*, each of which roughly halves the number of segments by merging them in pairs.

Specifically, at the point marked `# TODO: Task 3`, you should supply:

- A function `mergeRound(L,segs,M,key=lambda x:x)` that takes a list `L`, a division of it into non-overlapping segments (represented by a list `segs` of `Segment` objects), and a list `M` assumed to be of the same length as `L`. The function should merge the segments in pairs (i.e. merging `segs[0]` with `segs[1]`, `segs[2]` with

[1]This is exactly how the optional `key` argument works in Python's built-in `sort` method for lists — check out the online documentation.

segs[3], ...), filling up M from left to right with the results of the merges. If there are an odd number of segments, the last segment should simply be copied into the rightmost portion of M so that M is completely filled up.

Furthermore, your function mergeRound should return a list of Segment objects corresponding to the new division into segments within M. For instance, if segs consists of either 5 or 6 segments, your function should return a list of the 3 segments resulting from the merging/copying, ready for use in the next round.

To code all of this elegantly, you may wish to exploit the length information returned by mergeSegments and copySegment.

- A function mergeRounds(L,segs,M,key=lambda x:x) that repeatedly calls the above to perform several rounds of merging, alternately moving the data from L to M and back, until we're down to a single segment. You should ensure that at the end of the function call, the sorted items are stored in L, and your function should return L as its result.

  In the interests of space-efficiency, you should not use any scratch lists other than L and M for storing items, even temporarily.

The upshot of all this is that the provided function SimpleTimSort, in conjunction with your code, should behave as a general sorting operation for lists with respect to a given key function, using just a single scratch list of the same size as the input, plus relatively small amounts of other working data.

Marking scheme: Tasks 1,2,3 each carry 15 marks: 10 for correctness of your code, and 5 for the expected time- and space-efficiency properties. These will be assessed by an automarker supplemented by human eye inspection, and you will receive feedback on which tests your code passed or failed. Note that if your code is seriously incorrect, it may not be eligible for the efficiency marks.

The remaining 5 marks for Part A are for code style and clarity (e.g. comments, clear formatting, choice of variable names, avoidance of undue complication).

Part B: Space-efficient red-black trees [50 marks]

Before attempting this part, make sure you are familiar with the material on red-black trees from Lecture 9. Your task will be to provide implementations of the lookup and insert methods in a red-black tree implementation of dictionaries.

Take a look at the template file red_black.py. You will see a class Node for representing nodes of a tree: these carry key-value pairs as well as pointers to left and right children. (The special object None does duty for all trivial leaf nodes.) In contrast to most implementations of red-black trees, however, there is no pointer from a node to its parent. We are able to dispense with this (and so save some space) by implementing insert with the help of a *stack* that keeps track of the relevant path through the tree.

Type Node(5,'five') to the Python prompt. You'll see a readable representation of the node this has created, with the colour initialized to red. Note, however, that this representation isn't itself a valid Python expression that could be used to construct a node.

Near the end of the file, you will see some commands to construct a tree directly 'by hand'. Typing sampleTree to the prompt will reveal a (semi-)readable representation of this tree, with subtrees indicated by square brackets. (Once again, this isn't a valid Python expression.) Building trees directly in this way is useful for testing and debugging, but is of course 'unsafe' in that there's nothing here to enforce the rules for red-black trees. The safe way will be to start from the empty tree (created by `RedBlackTree()`) and repeatedly call the `insert` method, once you have implemented this.

You can also type

```
sampleTree.keysLtoR()
```

to compute a list of all the keys present, obtained by traversing the tree from left to right.

Within the class `RedBlackTree` itself, study the provided code for the methods `__repr__` (which creates the readable representation) and `keysLtoR`. These provide examples of methods implemented using a recursive helper function: this is the pattern you should follow in your implementation of `lookup` and `insert`.

Task 1 (6 marks). At the point marked `# TODO: Task 1`, implement a method

```
lookup(self,key)
```

that returns the value associated with the given key (or `None` if this key is not present). The use of a helper method is encouraged: you may follow the example of the `contains` method from lectures.

Task 2 (12 marks). At the point marked `# TODO: Task 2`, implement a method

```
plainInsert(self,key,value)
```

that inserts a new key-value pair into the ordered tree at the correct position, without worrying about colour information or the rules specific to red-black trees. The new node should be built using the `Node` constructor. If the given key is already present in the tree, the method should simply overwrite the existing value with the one given — in this case it should not create a new node or make any other change to the tree.

You may follow the example of the `insert` method for plain binary trees from lectures. The case of inserting into an initially empty tree will need special treatment.

Once you have got the basic form of this method working, you should add suitable commands so that the path from the root to the new (or updated) node is recorded as a list in the `self.stack` field. Specifically, after calling the method, this field should contain the list of `Node` objects from the root to the new node, alternating with strings `'l'` or `'r'` to indicate the left or right branches. (We take advantage of the fact that Python permits mixed-type lists.) For example, calling

```
sampleTree.plainInsert(5,'five')
sampleTree.showStack()
```

should result in the list

```
['2:two:B', 'r', '4:four:R', 'r', '6:six:B', 'l', '5:five:R']
```

If the tree is initially empty, the stack after the insertion should contain just one node.

Of course, the tree resulting from `plainInsert` may not be a legal red-black tree. The purpose of Tasks 3 and 4 is to do the fix-up needed to obtain a legal tree.

Task 3 (12 marks). Here the task is to write code to apply the *red-uncle rule* as many times as possible.

By the *current node*, we shall mean the one at the top (= right-hand end) of the stack. The red-uncle rule will be applicable if the current node, its parent and its uncle are all coloured red. In this situation, the rule may be applied by colour-flipping of nodes as described in the lecture.

Write a method `tryRedUncle(self)` that first tests whether the conditions of the rule apply. If they don't, the method should simply return `False` without making any changes. If they do, the method should apply the rule and return `True`. In the latter case, it should also remove items from the stack as appropriate (using the `pop` method for lists) so that the new current node will be the one you have just coloured red: this will set things up correctly for the next attempt at the red-uncle rule.

Here and in Task 4, you're encouraged to use the provided operations `opposite`, `getChild`, `setChild` to reduce needless duplication within your code.

Now write a method `repeatRedUncle(self)` that uses `tryRedUncle` to apply the rule as many times as possible (which may be zero times).

Task 4 (12 marks). Once we have applied red-uncle as much as possible, we will be in one of the *endgame scenarios* described in Lecture 9. Your main task here is to write a method `endgame(self)` that inspects the tree to see what needs to be done, and applies the appropriate action to turn it into a legal red-black tree. The provided functions `toNextBlackLevel` and `balancedTree` are there to help you in the case where some local re-balancing is called for. In other implementations (and in CLRS), this is achieved by means of certain *rotation* operations, but here I've followed the more simple-minded approach of inspecting the relevant subtree down to the next 'level of blacks', retrieving the relevant 7 components A,a,B,b,C,c,D (in the notation of the lecture slides), and building a new and more 'balanced' subtree from these.

Finally, add a method `insert(self,key,value)` that pulls everything together, inserting the given key-value pair into the tree and performing all necessary fix-up.

You should test your code thoroughly to ensure that you've handled all cases correctly. One way to test it would be using the provided function `TreeSort`, which uses a red-black tree to sort a given list in $O(n \lg n)$ time. This isn't a very serious contender for a sorting algorithm, though it's somewhat reminiscent of the way HeapSort works. You might also like to do some experiments to compare the efficiency of `TreeSort` with that of `SimpleTimSort` on lists of various kinds.

Marking scheme: Tasks 1,2,3,4 carry the numbers of marks stated above: all these marks will be for *correctness* of your code as assessed by the automarker. For these tasks, *any* reasonable correct implementation is likely to have the expected efficiency.

The remaining 8 marks for Part B are for code style and clarity — bearing in mind that needless inefficiency in your code may here be classed as 'bad style'!

Submission instructions: Please submit your completed files `timsort.py`, `red_black.py` (with these filenames) through the Learn assessment page. You may re-submit multiple times before the deadline — but if so, please re-submit **both files**, even if only one has changed. Otherwise, the other file won't be included in your submission.

Good luck!! — John