Informatics 2 – Introduction to Algorithms and Data Structures (2024-25)

Coursework 1: Smart Merge Sort and space-saving red-black trees

John Longley

October 16, 2024

This is a Python programming practical in two parts. In Part A, you will implement a version of Merge Sort with various enhancements (somewhat in the spirit of *TimSort*), and analyse its asymptotic runtime properties. 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. Part B (especially) will require a grasp of the organization of program memory as explained in Lecture 6. The purpose of the coursework is to develop and assess your understanding of the material from Lectures 1–10 and your ability to translate the ideas into Python code.

This is a credit-bearing coursework assignment worth 15% of the overall course mark. The deadline is **12 noon on Friday 8 November 2024**. Submission will be via **CodeGrade**, using the 'CodeGrade for IADS CW1' link within Learn. (*Do not use the submission mechanism of Learn itself, even though it provides one!*). You will be able to submit as many times as you like (the last submission made before the deadline will be the one that is marked). The coursework will be marked out of 100, using a combination of automarker tests on CodeGrade and inspection by a human marker. Your marks and feedback will be returned to you by Friday 29 November.

To get started, you'll need the template files `smartsort.py` and `redblack.py` and the supporting file `peekqueue.py`, all available from the Learn Assessment page. In each of the template 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 **70–80 lines** for each of the two parts. (Part A involves a bit less code than Part B, but also asks for some asymptotic analysis.) 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.

- The autotests will be run by CodeGrade using Python 3.12; in practice, any version of Python 3 will suffice for developing your code. For this coursework, you're *not* permitted to use any Python libraries other than those already imported by the provided code, nor to use any of Python's built-in sorting operations. The point in this coursework is to implement everything from the ground up.

- 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. The required function headers are included (as comments) in the template files. It's also important that you make no changes to the provided code, and add new code only at the points indicated.

  For each of the functions you implement, the automarker will run a number of tests to assess the correctness and (in Part A) time-efficiency of your code. The time tests won't be too stringent — any code that doesn't suffer from an obvious inefficiency should pass them, so no need to fine-tune your code for those last few percentage points in runtime. The human marker will then assess your code for space-efficiency and use of appropriate algorithms, and may adjust the automark either positively or negatively.

- For three of the functions you are asked to implement (namely `insertSort` and `merge` in Part A and `lookup` in Part B), you will be able to use CodeGrade to run some automated tests and see the results before you submit. (Once you've submitted your files to CodeGrade, select the 'AutoTest' tab.) This isn't intended to provide detailed feedback, but just to give you a general indication of whether you are on the right lines. (Note, however, that it's possible in theory for your code to pass these autotests but not to be eligible for full marks: e.g. if you 'cheat' by using a built-in sorting function, or if your code doesn't have the required space-efficiency properties, the code will pass the tests but the issue will be picked up by the human marker.)

- You should give some attention to matters of code style, as there will be a few marks allocated for this. In particular:

  - Your code should be *formatted* so that it is clear and easy to read. Some good advice on this is given in Python's PEP 8 style guide:

    `https://peps.python.org/pep-0008/`

    It's good practice to get into the habit of following these recommendations, though we won't mark you on whether you comply with them exactly.
  - All lines should be limited to a maximum of 79 characters.
  - Variable names should be sensibly chosen, and informative where possible, but remember that longer names can make the code hard to read overall. Short names are preferred for variables that are only used locally within a small region of the code (which will be the case for most or all of the variables you introduce).
  - You may add a modest number of **comments** to your code if they help to make it more readable. However, this should be done quite sparingly, as over-commenting can be counterproductive. 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. Where possible, it's best if the code is sufficiently clear that it speaks for itself.

  For some excellent guidance on these kinds of things, which will be equally important in *all* courses that involve programming, I recommend Perdita Stevens's book *How to write good programs* (Cambridge University Press), which includes examples in Java, Python and Haskell.

- It's your responsibility to **test** all of your code thoroughly before you submit it. You should expect to spend some time devising and constructing your own tests to ensure that your various functions work correctly in all cases — this is an important skill to develop. You may work on your own machine, but before submitting you should check that your code runs correctly on `python3.8` (as installed on the DICE machines).

- 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 too 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 a similar repository service while working on the coursework, please make *absolutely certain* that your files aren't visible to anyone else. Again, a breach of this may be treated as a case of academic misconduct.

Part A: An enhanced version of MergeSort [50 marks]

In this part, you will implement a version of MergeSort for sorting Python lists. Relative to the basic version described in lectures, this will have the following enhancements.

1. We will allow sorting using a binary *comparison operation* chosen by the user — which may be the usual Python `<=` operator but could be something else.

2. The implementation will switch to in-place InsertSort once the sublists it's working on become very short (recall that InsertSort is in practice faster than MergeSort for short lists).

3. The implementation will economize on space by using just a single scratch list of the same size as the input list. This is achieved via the method of Tutorial 2 Question 3.

4. Most interestingly, the implementation will detect, and take advantage of, significant portions of the input list that are already sorted (a common feature of real-world data). This means that our implementation will have best-case runtime of $\Theta(n)$, in contrast to the best case of $\Theta(n \lg n)$ for ordinary MergeSort. In this respect, our algorithm will be somewhat akin to *TimSort* and its successor *PowerSort* (as employed by Python's built-in `sort` operation), though we shall achieve this effect in a rather different way.

The implementation work is split into two tasks: Task 1 will achieve the first three of the above enhancements, and Task 2 will add the fourth. The result will be a sorting program that is fairly competitive given the choice of language and the amount of code involved.

Open the file `smartsort.py` and look at the code provided. You will see that three global variables are declared:

- `insertSortThreshold`: the maximum list size for which `insertSort` is used.
- `sortedRunThreshold`: the minimum size of the already sorted portions of the input that we bother to record.
- `comp`: the two-argument function to be used for comparing list items in the course of sorting.

In the provided code, each of these is initialized to a certain value (e.g. `comp` is initialized to have the same effect as Python's `<=`). However, bear in mind that your code may be tested with other values of these variables.

You are now ready to attempt the following programming tasks.

Task 1 (18 marks). In this task we implement, from scratch, a hybrid implementation of Insert Sort and Merge Sort. This part is intended to be fairly easy — you may draw here on code from lectures, tutorial sheets or textbooks, but will need to attend to a few details in order to comply exactly with the specifications given below.

At the point marked `# TODO: Task 1`, add code to define the following functions:

- (5 marks) A function `insertSort(A,m,n)` that uses *in-place InsertSort* to sort the portion of `A` from `A[m]` to `A[n-1]` inclusive. Your code should be modelled on the pseudocode for in-place InsertSort from Lecture 2, and should not create any other lists in memory. It's not permitted to call any built-in sorting function here!

  Your `insertSort` function should have the effect of sorting the required portion of `A`, but need not return a value. In the case `n<=m`, the function should return without doing anything to `A`.

  All comparisons between list items must be done using `comp`.
- (7 marks) A function `merge(C,D,m,p,n)` that takes two lists `C,D` and numbers `m <= p <= n`, where the portions `C[m],...,C[p-1]` and `C[p],...,C[n-1]` are assumed to be already sorted. (You may assume `C,D` each have length at least `n`.) The function should merge these segments and write the (sorted) result into `D[m],...,D[n-1]`. Again, you should not create any other lists in memory: in particular, *do not* use Python's list slice operation (e.g. `C[m:p]`) as this creates a new list! Your function need not return a value.

  Again, all comparisons between list items must be done using `comp`.
- (6 marks) A function `greenMergeSort(A,B,m,n)` (so called because it doesn't create extra garbage) which sorts the portion of `A` from `m` to `n-1` inclusive, putting the result in this same portion of `A`. You may use the list `B` as scratch space, but your code should not create any further lists. Your function should be modelled on the MergeSort implementation given in the sample solution to Tutorial 2 Question 3 (available from the course schedule page), with its four-way recursive splitting. However, you should use your function `merge(C,D,m,p,n)` for all the necessary merges. As a base case, you should call your `insertSort` function if the portion of `A` to be sorted has length `insertSortThreshold` or less.

Now look at the provided function `greenMergeSortAll`. You should check that this function, in conjunction with the code you have written, behaves as a reasonably efficient sorting operation, say for lists of length $\leq 1,000,000$.

Task 2 (16 marks). We now develop some code for detecting already sorted portions of the input list, storing their positions in a queue, and using this information to optimize the execution of our MergeSort algorithm. This part may involve a little more thought.

You should start by looking at the implementation of the class `PeekQueue` in the file `peekqueue.py` — this is similar to the queue implementation that featured in Python Lab Sheet 3, but is equipped with a `peek` method for inspecting the next queue item without removing it. This is the class that you should use for the queue you build.

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

- (6 marks) A function `allSortedRuns(A)` which identifies all (maximal) already sorted segments of `A` of length at least `sortedRunThreshold`, and builds and returns a queue recording their positions. A sorted segment from `A[i]` to `A[j-1]` inclusive should be represented by the pair `(i,j)`. E.g. if `sortedRunThreshold` is set to 3 ,and `A` is the list

  ```
  [5,2,3,3,4,1,10,8,11,12]
  ```

  then `allSortedRuns(A)` should return a queue `Q` containing (in order) the pairs `(1,5), (7,10)`, corresponding to the fact that the segments `A[1:5], A[7:10]` are already sorted. (Note that `A[5:7]` is also already sorted, but is not long enough to qualify for inclusion.)

  As in Task 1, you should use the function `comp` for all item comparisons. You should also ensure that your implementation avoids needless inefficiency. (As a hint, my own implementation involves two nested while-loops.)

- (6 marks) A function `isWithinRun(Q,i,j)` that returns `True` or `False` according to whether the portion `A[i],...,A[j-1]` is already sorted according to the information in `Q` (that is, whether this portion is a sub-portion of one of the sorted portions recorded in `Q`). You should give some thought to how much of the queue needs to be inspected to achieve this in a clean way. You may assume that if `isWithinRun` is called with arguments `Q,i,j`, it will never be later called with arguments `Q,i1,j1` where `i1<i` — though it may be later called with the same `i` and a different `j`.

  You don't need to write much code to achieve this (my own version is 6 lines including the function header), but it may require some thought, and a brief comment to explain how your code works may be in order.

- (2 marks) A function `smartMergeSort(A,B,Q,m,n)`, similar to `greenMergeSort`, but which starts by checking whether the portion `A[m],...,A[n-1]` is already sorted according to the information in `Q`, and if so does nothing. (Here you may simply copy-paste your code for `greenMergeSort` and then make the required changes.)[1]

[1] There's a further optimization one could imagine, but which we won't attempt to incorporate here. What if, say, `A[m],...,A[n-1]` wasn't already sorted, but at least the first half was? Then we could save ourselves the work of merging the first two quarters. You might enjoy thinking about how this could be implemented, but don't try to do this in your code.

Further 2 marks: The upshot of all this is that the given function `smartMergeSortAll`, in conjunction with your code, should behave competitively as a general sorting operation for lists with respect to the comparison operation currently stored in `comp`.

Task 3 (10 marks) In this task, you are not required to write any further code, but rather to provide some asymptotic analysis of the runtime properties of `smartMergeSortAll`.

1. (5 marks) Explain why the worst-case runtime for `smartMergeSortAll` on input lists of length n is $O(n \lg n)$. For the analysis of the main recursion, you should apply the Master Theorem from Lecture 10. You should also carefully account for the runtime of all other tasks involved.
2. (5 marks) Let's say a list is *nearly-sorted* if at most one item is in the wrong place. (More precisely, a list is nearly-sorted if it can be turned into a sorted list by deleting just one item.) What is the asymptotic worst-case runtime of `smartMergeSortAll` on nearly-sorted lists of length n? Justify your answer as carefully as you can.

Include your answers as Python comments at the point marked `# TODO: Task 3`. You should write a maximum of 15 lines of plaintext for each of the above questions. Any readable way of representing mathematical notation within plaintext will be acceptable (e.g. `Theta(n), 2^k`).

The final 6 marks for Part A will be awarded by the human marker for code clarity and style (good structure, formatting, choice of names, commenting etc.) as described on page 2 of these instructions.

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 `redblack.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 paths 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. The new node should be built using the `Node` constructor, which will initially colour it `Red` (and you should leave it like this). 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 the special values `Left` and `Right` to indicate the branches chosen. (These are the possible values for the Python *enumeration type* `Dir`, which is declared by the provided code.) 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']
```

Here, for convenience, the values `Left` and `Right` are *displayed as* the strings `'l'` and `'r'`, but the values that your code puts in the stack should be the actual `Left` and `Right` as declared by the provided code. 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 some 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. Colours should always be represented by the values `Red`, `Black` of the provided enumeration type `Dir`.

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. (You may assume that the stack is in the state in which is has been left by calling `repeatRedUncle`.)

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 e.g. 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. Some inspection of the provided code may help you to understand what is going on.

Now 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.

The final 8 marks for Part B will be awarded by the human marker for efficiency and code style. (Not that the autotests for Part B will not test for efficiency. However, in this case, it's likely that any reasonable correct implementation will have the efficiency we expect.)

You should test your code thoroughly to ensure that you've handled all cases correctly. One way to test it (though don't rely exclusively on this) 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 `smartMergeSortAll` on lists of various kinds.

Submission instructions will be announced by email by Friday 27 October.

Good luck!! — John