Informatics 2 – Introduction to Algorithms and Data Structures (2025-26)

Coursework 1: Smart Mergesort and Perfect Hashing

John Longley

October 17, 2025

This is a Python programming practical in two independent 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 a scheme for perfect hashing for a given set of keys, giving a flavour of state-of-the-art work in this area, and use it to create an efficient dictionary implementation. 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 20% of the overall course mark. The deadline is **12 noon on Friday 7 November 2025**. The coursework will be marked out of 100, using a combination of automarker tests on Gradescope and inspection by a human marker. Your marks and feedback will be returned to you by Friday 28 November.

To get started, you'll need the template files `smartsort.py` and `perfhash.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 function or method definitions at the points marked `# TODO`.

Some general points:

- The total volume of Python code you're expected to write is not vast. My own implementation has around 70 lines for each part, excluding comments. Don't worry if your version is a little longer or shorter than this — but if you find yourself writing twice as much, you should pause to ask yourself whether there's a simpler approach.
- The autotests will be run by Gradescope using a current version of Python 3; in practice, any Python 3 version will suffice for developing your code. For this coursework, you're *not* permitted to import any Python libraries other than those already imported by the provided code. 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 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 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 Gradescope to run some automated tests and see the results before you submit (see the end of this document for guidance). 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 that it's possible in theory for your code to pass these autotests but not to be eligible for full marks, if the human marker spots some issue not picked up by the autotests.)

- 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 should add a modest number of **short comments** to your code where 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 sorts of things, which will be equally important in *all* courses that involve programming, we 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. Before submitting, it's worth checking that your code runs correctly on Python as installed on the DICE machines (the autograding will actually be done on Gradescope's machines using a similar setup). As a bare minimum, please ensure that your files load correctly into Python without any syntax errors — if they don't, we won't even be able to run the autotests and *very few* marks will be available.[1]

[1]A common source of problems in Python coursework is incorrect indentation. E.g. a method that's supposed to be part of a class definition may fail to be treated as such if the indentation indicates that the class definition has already finished. This may be something to double-check in case of problems.

- 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, we will be running **drop-in lab sessions** in Weeks 7 and 8 (times and venues are advertised on Open Course), and the demonstrators there should be able to give you one-to-one assistance. 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 and answered! 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.

  It is similarly an offence to use AI systems or any other online tools to write *any part* of your solutions for you.

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.

For Part A, it is forbidden to use the Python `sort` method or any other built-in sorting operations — the point is to see for yourself how such operations may be implemented.

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. You should interact with the `PeekQueue` class only through the constructor and methods provided — your code should not rely on the concrete representation this class uses, and should work equally well if a different implementation of peek queues were substituted.

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.)[2]

Further 3 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 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 rendering mathematical notation within plaintext will be acceptable (e.g. `Theta(n), 2^k`).

The final 5 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: A modern perfect hashing algorithm [50 marks]

As explained in Lecture 8, a *perfect hash* function for a fixed set S of keys is a function $\# : S \to \{0, \ldots, m-1\}$ (for some chosen m) such that no two keys map to the same integer. A perfect hash function $\#$ is *minimal* if $m = |S|$, so that $\#$ is actually a bijection $S \to \{0, \ldots, m-1\}$.der If we are able to find a perfect hash function for a given set of keys, we can implement a hash table for these keys with fast worst-case lookup. Such schemes (including minimal ones) have found practical applications in databases, computer networks, compressed text indexing, language models, and other areas.

The method we follow is due to Belazzougui, Botelho and Dietzfelbinger (2009), and enables us to construct perfect hash functions that can be represented very compactly in memory. A similar approach (with bells and whistles) is also employed by recent state-of-the-art work in the area.[3] The instructions below give all the information you will need, but the file `BBDslides.pdf`, available from Learn, may serve as a useful supplement (including some pictures).

[2]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.

[3]See for example G.E. Pibiri and R. Trani, *PTHash: Revisiting FCH Minimal Perfect Hashing*, SIGIR 2021.

Take a look at the template file `perfhash.py`. There is some simple machinery for hashing of strings over `a..z` modulo a number `p`. The precise details aren't important, but you should experiment a bit with the function `modHash` to see what it does. There is then a function `miniHash` which, for a given number `m`, yields a whole family of (reasonably 'independent') hash functions mapping strings to integers `0,...,m-1`. For example:

```
miniHash(1000,0)('aardvarks')
miniHash(1000,1)('aardvarks')
```

Our goal is to construct a *perfect hash function* mapping a specified list of `n` English words (given by `keys`) to integers `0,...,m-1` for some specified `m`. (The intention is that `m >= n`, but that `m` is not *much* larger than `n`.) The method proceeds as follows.

First, we 'mod hash' our set of keys down to numbers `0,...,r-1`, where `r` is some suitable prime number (chosen by code later in the file). This results in a list `L` of `r` *buckets*, where each bucket is itself a (possibly empty) list of strings. This is the purpose of Task 1 below.

Next, we try to find for each of these buckets a number `j` such that `miniHash(m,j)` is a suitable mini-hash function to apply to the strings in that bucket. 'Suitable' means that all of these mini-hash functions *taken together* should map the contents of all buckets to codes `0,...,m-1` with no clashes.

Your main challenge (Task 2) will be to write a function `computeMiniHashIndices(L,m)` that takes a list `L` of buckets (that is, a list of lists of strings over `a..z`) and a desired table size `m`, and suitably selects a number `j` for each bucket as above, storing the number `j` for the `i`th bucket at position `i` in a list `R`, whose length is `r`, the number of buckets (i.e. the length of `L`). This list `R` is what your function should return; it will become the value of the field `hashChoices` in the `Hasher` class. The `hash` method for this class then gives the hash function we have constructed, which is intended to be a perfect hash function.

How do we go about choosing `j`-values to ensure this? We proceed as follows:

- First, sort the list of buckets `L` in order of decreasing size. Within the sorted list, we will want each bucket to be paired with a number `i` indicating the position in the original list `L` where this bucket came from: thus, each entry in the sorted list will be a pair `(i,L[i])` for some `i`.
- Create a list `T` of `m` booleans, initially all `False`, intended to record which slots in the main table are currently 'taken'. Also create a list `R` of integers (initially zero), one for each bucket.
- Go through the buckets in order of decreasing size. For each bucket `B`, search for the smallest integer `j >= 0` such that the mini-hash function `miniHash(m,j)` is *suitable* for `B`. This means that:
  - it maps all elements of `B` to currently free slots in the main table,
  - it doesn't map any two elements of `B` to the same slot.

  Once a suitable value of `j` is found,[4] record this value as `R[i]` (where `i` is the original position of the bucket `B`), and mark all the slots to which we are sending elements of `B` as 'taken'.

[4]Actually, there's no absolute guarantee that a suitable `j` will *ever* be found. But as long as our mini-hash functions are 'sufficiently random and independent', this will happen 'almost certainly', and the approach works extremely well in practice. You need not worry about the possibility that the search will fail.

The provided class `Hasher` will call your code to create a perfect hash table. Task 3 will then use this to create a dictionary implementation with fast lookup.

Task 1 (6 marks). The first task is to write some simple code for constructing classic bucket-array hash tables. At the point marked `# TODO: Task 1`, provide implementations for the following two functions.

- (4 marks.) A function `buildHashTable` which, given a list of keys `L`, a desired table size `r` and a hash function `h` mapping the keys to integers `0,..,r-1`, returns the corresponding list of `r` buckets, with the bucket for each hash code `c` appearing at position index `c`. Each bucket should itself be a list of keys (possibly empty).
- (2 marks.) A function `buildModHashTable` that calls the above to create a hash table using the `modHash` function (as defined earlier in the file) with a given modulus `p`. (Hint: use Python's `lambda` construct.) Again, this should return a list of buckets as above.

Task 2 (24 marks). At `# TODO: Task 2`, implement `computeMiniHashIndices(L,m)` as described above, returning a list of mini-hash indices, one for each bucket, in the order in which the buckets appear in `L`. Although the task may seem complicated, the list comprehension syntax of Python allows it to be coded quite economically. (As a rough guide, my own implementation takes 26 lines including comments.) You may define any helper functions you find useful (although my implementation doesn't have any).

In this part, you *are* permitted to use Python's built-in `sort` method, and this may be helpful — consult the Python Standard Library documentation for details.

You will now be able to construct perfect hash tables as follows. Set `keys` to be a list of English words, for example this one:

```
https://www.freescrabbledictionary.com/enable/download/enable.txt
```

(Such a list can be loaded in using the `wordlist` function defined at the end of the template file). Then type e.g.

```
H = Hasher (keys,5.0,0.8)
```

This should succeed within a few seconds, using your code to build a data structure `H` which can then be used for fast hashing (e.g. `H.hash('aardvarks')`). The parameter 5.0 approximately determines the average bucket size used (via `r` $\simeq$ `n/5.0`): the higher this parameter, the more compact the resulting data structure `H`, but the longer it may take to find suitable indices.[5] The parameter 0.8 gives the load for the resulting hash function (so that `m = n/0.8`); again, the closer this parameter is to 1.0, the less space we waste, but the harder our Hasher will be to construct. Taking 1.0 for this parameter gives us a *minimal* perfect hashing — this should succeed within a few seconds with an average bucket size of around 3.0, but it may struggle with higher bucket sizes.

[5]For reasonable parameter choices, the space occupied by `H` will not exceed 3 or 4 bits per key (independently of the sizes of the keys)! This is obviously tiny compared with the keys themselves, let alone any values we might associate with them.

It's not too hard to check that your `H` really is a perfect hasher by using the provided function `buildHashTable` to build an ordinary bucket-list hash table from it, then checking that this contains no buckets of length > 1. This will entail writing a small amount of test code and we recommend that you try this, but please do not submit this extra code.

Marking for Task 2: 16 marks for correctness and 8 marks for efficiency. As stated in the preamble, any correct implementation that avoids blatant inefficiencies will qualify for full marks.

Task 3 (6 marks). Later in the file, you will find a template for a class `HashDict` which uses minimal perfect hashing to provide a dictionary implementation with fast lookup. The constructor, which is already provided, creates such a dictionary from a given set of key-value pairs, subject to some parameters. Study this to understand how it works. Note that each key-value pair should be supplied as a list `[k,v]` rather than a tuple `(k,v)` — this is so that we can later update the `v` component if we wish.

Your task here is to provide implementations of the following two methods, at the point marked `# TODO: Task 3`.

- (3 marks.) `lookup(self,w)`: Returns the value associated with the key `w`. Should return `None` if `w` is not present in the dictionary, or if `None` is associated with `w`.
- (3 marks.) `setValue(self,w,v)`: Updates the value associated with `w` (if present) to `v`. Should return `True` if `w` is present and the update succeeds, or `False` if `w` is not present.

Both implementations should run in $\Theta(1)$ time, i.e. within a time bound independent of the number of keys.

You should devise a way to test the resulting `HashDict` implementation on a suitable list of key-value pairs. Again, you need not submit your test code.

Task 4 (9 marks). The last task is more open-ended. You are invited to consider what it would take to add an `insert` method for adding a new key-value pair to the dictionary (where the key is not one already present). This will inevitably involve some costly work in some cases, but you should think about how the problem might be mitigated.

Your answer may consist *either* of English text (as comments) explaining a strategy for approaching this, *or* of a working Python implementation of an `insert` method (with the method header provided), or a mixture of both. A solution scoring full marks will most likely consist of a simple Python implementation along with a comment on how one might do better. Helper functions are permitted. Whatever your choice, your solution should take *no more than 30 lines, max 79 characters per line* (not counting blank lines).

The final 5 marks for Part B will be awarded for code clarity and style as in Part A.

Submission and autotest instructions

You should submit your work to **Gradescope**, using the Assessment > Coursework 1 > Gradescope link within the Inf2-IADS Learn page. Gradescope will be open for submission **from Friday 24 October**. Please submit your two completed files under the original filenames — `smartsort.py` and `perfhash.py` — and do not attempt to submit any other files.

Immediately after you have submitted, the autograder will take around 30 seconds to run a few correctness tests on your implementations of `insertSort` and `merge`. You will then see which tests passed and which failed, and you may use this information to revise your code if necessary. (These tests are a selection of those that will be used for the actual marking.) This is intended just to provide some initial reassurance that you are on the right track — we intentionally do not provide such visible for the remainder of the coursework, as it is important that you learn to devise ways of testing code for yourself, rather than being reliant on someone else to do this.

The autotests will also let you know whether you have submitted both files under the correct names.

You can re-submit as many times as you like up to the deadline — the last submission made before the deadline will be the one that is marked. **However,** if you do this, please re-submit **both files**, not just the one that has changed! Only the files included in your most recent submission will be marked.

Good luck!! — John