STAT0041 Algorithms and Data Structures: ICA 2022

Instructions

The deadline is 9th of December, 5pm, via Moodle. Please see Moodle instructions for submission.

This ICA must be done in pairs. If you have problems finding a pair, contact me.

The first question is worth 20% of the total marks. The other two, 40% each.

You can write solutions in either Python or R, or a combination of both for *different* questions. No single question should be written in two different languages!

In my instructions, I always refer to arrays starting from index 1. If you are using Python, feel free to make them always start from index 0. This is not an important point. Just be coherent.

Commenting code is recommended, but keep it light. If you think particular pieces of code are clear enough, no need to clutter them with redudant comments. Your future co-workers will thank you for your habit of keeping code uncluttered.

No need to worry about language-specific efficiency, e.g. both Python and R are terrible with loops, but let's not worry about this. Programming language pyrotechnics is not the point of *STAT0041*.

It's OK to deliver questions in separate files, `.py` or `.R`, but no more than one file per question! Jupyter notebooks or R Markdown are especially welcome (if you follow this route, please just create one file for all answers). You can only upload one file, so you need to zip whatever files you have. No unusual compression formats like RAR please! Zip only.

Any questions where you need to write in plain English must be done as comments/Markdown text in the respective `.py` / `.R` / `.ipynb` / `.Rmd` files. Indicate very clearly which answer corresponds to which question.

Despite the verbosity of each question (needed to remove ambiguities), rest assured that each deliverable amounts to actually fairly little coding. Expect that your full solution for each question will be shorter than the corresponding question setup!

The deadline is 9th of December, 5pm, via Moodle. Please see Moodle instructions for submission.

Question 1

Calculating order statistics is a very standard task in descriptive statistics. Let's investigate empirically how much of a cost it is to use sorting as a way of calculating them.

Long story short: you will investigate how long it takes to compute order statistics using sorting, if every now and then you need to sort the data. This will be done by implementing a pretend simulation of changes to the data that force a re-sorting. You will compare your implementation against whatever built-in way of finding medians R or Python has to offer.

Here are the details. At this end of this question there are several suggestions on how to code some of the finer details in R or Python.

Deliverable 1: Write a function `simulate_order_sort(m, n, p)` to simulate requests for order statistics as if they were coming from real people. Argument `m` indicates that we are going to run `m` simulated requests for order statistics. Argument `n` indicates that we are going to simulate arrays of length `n` from which order statistics will come: this will be done by sorting the data and picking the corresponding position `o` in the sorted data, where `o` is chosen randomly in $1, 2, 3, \ldots, n$. Argument `p` is a number between 0 and 1 that controls how often we need to re-sort the data because of pretend changes to it.

The simulator must work as follows:

1. Simulate an array `x` of length `n`, each entry drawn independently from a standard Gaussian.
2. Sort `x`.
3. Create an array `y` of length `m` that will store results.

4. Repeat the following `m` times. First, draw an integer `o` in $\{1, 2, \ldots, n\}$ uniformly at random (assuming `x` starts from index 1), and copy `x[o]` to `y[i]` at the corresponding iteration number `i` in $1, 2, 3, \ldots, m$. As seen in Chapter 2, `x[o]` will be the order statistic corresponding to the request `o`. At the end of the iteration, with probability `p` you will change the data: this means drawing `x` again and re-sorting it (therefore, with probability $1 - p$ there is no change of data and no re-sorting).
5. Add the end of the `m` iterations, return `y`.

Deliverable 2: Write a short script to test your `simulate_order_sort` function. For some m (say, $m = 10000$) and n (say, $n = 10000$ also), choose a range of values for `p` (say, $\{0, 0.01, 0.1, 0.5\}$) and print the wall-clock time required to run `simulate_order_sort` for each choice of `p`. Follow up the code by writing a sentence or two explaining where the variability in wall-clock time caused by `p` comes from.

Deliverable 3: Say all we care about is the median (here defined as the order statistic at position $\lceil n/2 \rceil$). Create a modified version of `simulate_order_sort`, to be called `simulate_median_sort`, where the only difference is that `o` is always set to $\lceil n/2 \rceil$. Write a second function, `simulate_median(m, n, p)`, with analogous inputs but which does no sorting at all: all medians are calculated by some existing choice of median computation function you can choose from R or Python. Do the wall-clock time analysis for both functions at different choices of `p`. Report the differences you see and whether they correspond to your expectations.

Programming hints:

- As we have seen in the notes/slides, for any z we have that $\lceil z \rceil$ is the "rounded-up" version of z. This corresponds to functions like `ceiling` in R or `numpy.ceil` in Python.
- If you are answering this question in Python, it's OK to use Python lists instead of a "proper" array of the likes you would see in NumPy. Feel free to use arrays/lists which start at 0, but adapt the range of `o` accordingly.
- Several exercises in the Exercise Sheets do wall-clock computations and simulated data. For instance, see Exercise 8 of Exercise Sheet #1. In R, function `Sys.time()` has a similar functionality.
- To sample from a set of integers in Python, you can look at the NumPy function `np.random.choice`, which by the way was used in the solution of Exercise 2 from Exercise Sheet #2. In R, function `sample` has a similar functionality. The same functions can be used to simulate a coin flip with probability `p`, which you need in order to decide whether it's time to pretend that the data has changed. For standard Gaussians, `np.random.normal` in Python and `rnorm` in R is what you will look for.
- To calculate medians in Python, feel free to use function `median` from the package `statistics`. In R, function `median` is built-in already.

Question 2

*You will implement a set of functions that manipulate an array-like data structure. Let's call this data structure a "stretchable array" (*s-array*), for the lack of a better name. The idea is to build upon a standard array, but delivering a data structure that allows for extending its size in a way that the user of such a data structure doesn't need to know.*

Long story short: *every time we modify an element that is outside the range of the array, we will double the array in size. You will write the record and relevant functions that manipulate this data structure.*

Here are the details. They are long and somewhat tedious in order to make the question as clear as possible. Your solution will probably be shorter than the framing of the question...

Background: A variable of "type" *s-array* is a record which we will describe shortly. We will assume that the data in a *s-array* are always integers. We will operate on a *s-array* only via the following functions:

- `s_array_create(l)`. Creates a *s-array* of *capacity* at least `l` with all entries initialised to `0`. It should return a new record of "type" *s-array*. The definition of "capacity" is given below. The idea of "at least `l`" will also be explained below.
- `s_array_set(x, i, value)`. Set the i-th position of *s-array* `x` to `value`. Informally, this is what we would normally code as `x[i] = value` in Python or R. But `x` is not meant to be a standard built-in array that comes pre-defined in

the language. To avoid fiddling with Python's or R's syntax, all operations in a *s-array* will be done by explicit functions. This function doesn't need to return anything (but see the very bottom of the **Programming hints** section if you are using R).

- `s_array_get(x, i)`. This function returns the value stored at the i-th position of *s-array* `x`. Informally, this is equivalent to returning `x[i]` if we were to use a friendlier syntax. This will be a trivial function to implement, as you should quickly realise from the specification below.
- `s_array_len(x)`. This function returns the attribute `length` of `x`, as described below.

A *s-array* must be implemented as a record with the following attributes.

- `data`: an internal standard array where the data is actually stored. This means that e.g. `s_array_get(x, i)` should, under the hood, return `x.data[i]`.
- `capacity`: the length of `data`.
- `length`: that's where things get interesting. Here, `length` is the highest index among all elements which have been modified by a call to `s_array_set`.

What's this weird business about the difference between `capacity` and `length`? The idea is that we are allowed to set the value of an entry which lies beyond the size of `data`. Since `data` is an array that is assumed to be of fixed size, this will mean creating a new, larger, array and destroying the old, while copying information. In particular, everytime there is a call `s_array_set(x, i, value)` where `i > x.capacity` then, inside the `s_array_set` function, the following should happen:

1. We create a new array `new_data` that has *twice* as many elements as `data`, setting all of its entries to zero.
2. We copy all entries of `x.data` to the corresponding positions of `new_data`.
3. We set `x.data` to now point to `new_data`.
4. We update `x.capacity` and `x.length` accordingly (the former is just the size of `data`, the latter is just `i` *if* we start counting positions from 1 instead of 0).

The idea is, if we want to print the entries of some *s-array* `x`, we stop at the last position that we know to be non-zero. We will treat zeroes as a special, "uninteresting", value. Given this explanation, your functions must also do the following:

- If we call `s_array_set(x, i, value)` where `i` exceeds the length but not the capacity of `x`, then all we do is to update the length of `x` to indicate where the highest-indexed modified element can be found.

For what follows, you will provide code in either R or Python. For instance, step 3 above can be done in Python and R by simply assigning an array (or list) to another. A *record* in R can be implemented just as a list with named fields (you do need to name the fields, e.g. you want your code to read as `x$data` as opposed to `x[[1]]`, for instance). A record in Python can be implemented as a class. A "*standard array*" in Python can be just a Python list. No need to invoke NumPy or any other special data structures for this question.

Deliverable 1: Write all functions above, adding some basic comments to the code to explain the logic of what you are doing. A *special requirement* is the following: assume that you have a project manager that tells you that the length of the `data` attribute should always be a power of 2, with minimal waste. That is, a call that creates a *s-array* by, say, `x = s_array_create(12)` should create a record where attribute `data` is an array of length 16, which is the smallest power of 2 greater than or equal to 12. A call to `s_array_create(17)` should create an array `data` of size 32 and so on. The `12` in the call `x = s_array_create(12)` just means that `x.capacity` must be *at least* 12, not exactly 12.

(This may sound arbitrary, but it has a reason: this is because the hypothetical manager believes that empirically, given the experience they have, this will provide a good trade-off between memory use and multiple array destruction/copying. Such trade-offs are not that uncommon when engineering a data structure.)

Deliverable 2: Write a short explanation (< 200 words) of what you personally think are the good and bad properties of the above implementation when the goal is to have a data structure where: (i) most of the entries are 0s and are not interesting; (ii) the size of the data structure may change as we use it. Briefly compare to some alternative way, of your choice, of implementing such a structurethat is not based on standard arrays but another starting point like a linked list.

Deliverable 3: Say that from time to time you may want to check whether you could shrink the physical length of the `data` attribute of a particular *s-array* `x` (as some non-zero values of `data` get turned to 0 by `s_array_set` calls). Implement a function `s_array_pack(x)` which checks the position of the last entry of `x.data` that is not zero (we assume that 0s mean "irrelevant"). We then check whether it is possible to resize `x.data` to a smaller power of 2 size while not losing any

"relevant" data, and do the reduction. For example: if `x.data` is an array of size 64 (`x.capacity == 64`) and the non-zero element of `x.data` of highest index is `x[30]` (`x.length == 30` , assuming we start counting from 1), then we can update `x.data` to a size 32 array, updating `x.capacity` to 32, accordingly.

Deliverable 4: Provide a short criticism (< 200 words) of your implementation of `s_array_pack` , and how you could improve it for a future version (no need to write any further code!). Or, if you think it is perfect, you can make the case of why you think so!

Deliverable 5: Write a short piece of code demonstrating the functionality of a *s-array*, in any way you want. Feel free to be minimalist. Just make sure that each function above is called at least once. The main point of this question is to force you to test your code.

Programming hints:

- check out how logarithms can be calculated in R or Python so that you know how to "round up" numbers to the closest power of 2 (for instance, $log_2(12) \approx 3.58 < 4$, which means that $2^4 = 16$ is the capacity that should be allocated initially). So basically you need something like $2^{(\lceil \log_2(l) \rceil)}$ to get the smallest power of 2 no smaller than l. Look for functions like `ceil` or `ceiling` , `pow` or `^` , `numpy.log2` or just `log2` , depending on your choice of language. Make sure the result is converted to an integer in the end (`int(<number>)` in Python or `round(<number>)` in R etc.)
- You can make your *s-array* start at index 0 or index 1, it's up to you (probably whatever it's easier in your doing it in Python or R). Bear in mind that if you start at 0, you may want to think a bit what `x.length` and `x.capacity` should be...
- If you are implementing this in R, you may have difficulties with functions that modify the contents of an array: R's default behaviour is "passing by value". Unfortunately there is no elegant way out (if you find one that doesn't require obnoxious overuses of object-oriented programming, let me know!). You may need to use a function like `s_array_set(x, i, value)` by ridiculously returning the entire array back, resulting in calls like `x <- s_array_set(x, i, value)` . R wasn't created having efficiency in mind, but readability, and there is something positive to be said about avoiding calls by reference. Still, don't shoot the messenger...

Question 3

Graph traversal algorithms like breadth-first (BFS) and depth-first (DFS) search are among the most common framings of search problems that exist in practice. How they perform empirically may depend on the type of problem and how they are coded, as in the worst-case scenario they are equivalent.

Long story short: *you will analyse empirically how BFS and DFS work across a variety of simulated problems and metrics.*

Here are the details.

Background: You will use the package *igraph*, which exists for both Python and R (find installation instructions at https://igraph.org). Here's the Python documentation and here's the R documentation. Both Python and R users may benefit from the Python tutorial, keeping in mind that the syntax will be different in R.

Part of the idea of this question is that you will be (partially) left to your own wits on how to understand a package documentation based on your knowledge of algorithms and data structures (without which the documentation would look like gibberish). That's what you should expect in your professional life! But I know your time is limited, so I'll point out some of the functions you will need.

There are different ways graphs can be represented in *igraph*. I'd recommend you to first browse the *Graph generation* section of the documentation. It suffices the understand that we can use adjacency matrices and adjacency lists to represent a graph.

Deliverable 1: Write a function, to be called `search_eval(m, n, p)` , which runs `m` simulations as follows. In each simulation, this function generates a random directed graph over `n` vertices. At the end of each simulation, we report two numbers: the wall-clock time of running BFS and DFS on each of the `m` cases. So, `search_eval(m, n, p)` must return a

$m \times 2$ matrix `r`, where the first column of `r` are the m wall-clock times for BFS, and the second column are the corresponding times for DFS.

To generate a random graph, I suggest using an adjacency matrix (say, called `g`) for representation, and independently setting each entry `g[i, j]` to 1 with probability `p` (the third argument in the call fo `search_eval`), 0 otherwise. You can implement this in a variety of ways using random number generators of your choice, with or without loops, even in NumPy, but you should do this *without* making using of any specific *igraph* function that generates random graphs.

For each simulation, your function must choose one vertex uniformly at random to be a "source" (called a "root", by *igraph*'s documentation) vertex from which a search starts. Your function must then run BFS and DFS using *igraph*, recording the wall-clock time separately for each. See Exercise 8 from Exercise Sheet #1 for an example of wall-clock time experiment. There is no need for a "destination" vertex. When we run a BFS or a DFS in a graph using *igraph*, it just calculates how many steps it takes to reach all other vertices (if at all reachable).

The function for depth-first search is called `igraph_dfs` in R, and just `dfs` in Python. You should figure it out the details on how to use them from the *igraph* docs (you can just a web search to find the corresponding pages, for instance). You are expected to be more independent here, as you would if someone assigned you this task as part of a job.

Deliverable 2: In this question, we will test whether there is any empirical difference in the DFS and BFS implementations in *igraph* for your randomly generated graphs. You can think of this as a sanity check of this library. We will do this in two ways.

First, write a script that calls `r = search_eval(m, n, p)` with `m = 1000`, `n = 200` and a variety of choices of `p` (I suggest varying from small numbers, like `0.01` to up to `0.1`. It's up to you.). For each instance of `p`, do a *z-test* to whether the difference between the first and second columns of `r` comes from a distribution with zero mean. The hypothesis is rejected if the resulting p-value is less than a threshold (say, 0.05).

Second, plot either a scatterplot of a qqplot of the two columns of `r` to visually judge how different (if at all) the run-time of BFS and DFS in *igraph* may look to you. Write a sentence or two with your interpretation.

Deliverable 3: A call to BFS or DFS in *igraph* returns an array (among other outputs, read the docs) indicating the ordering by which each vertex was found in our search (if they were found at all, as there might be no path linking the source to a particular vertex). Let's call this array `order` from now on. This means that the DFS/BFS implementation of *igraph* is meant to list how many steps it takes to go to all vertices from a particular source vertex, according to the respective search schedules of each algorithm.

Let's create a family of graphs to empirically see how it affects DFS vs BFS. You will write a function `depth_charge(m, w, d)`. This function will create m simulations of a directed graph defined as follows. Each graph has a "root" vertex v_0, and w chidren $v_1, v_2, \ldots, v_w$. We then choose one v_i of $v_1, v_2, \ldots, v_w$, uniformly at random, to start a chain $v_i \to v_{w+1} \to v_{w+2} \to \cdots \to v_{w+d}$. We then call BFS and DFS on this graph, and peek into output `order` to see how many search steps it took to find v_{w+d} starting from v_0. We store this information for all m trials and return a $m \times 2$ matrix, similarly to what you did in Deliverable 2.

Given your code for `depth_charge`, write a script that calls it for `m = 1000`, `w = 100` and `d = 100`. Do a comparison between *igraph*'s implementation of BFS and DFS by plotting a histogram of the first and second columns of the output `r = depth_charge(1000, 100, 100)`.

Redo the analysis above, using a modified version of `depth_charge` (call it `depth_charge_1`) which always chooses v_i to be v_1. Again, do the same by setting v_i to be v_w, call that `depth_charge_final`.

What are your conclusions, particularly in light of the following conjecture?

- a standard implementation of BFS or DFS would treat the ordering of the children of v_0 in whatever order you entered them when creating the graph variable. That is, if you created a adjacency matrix, then the children of a vertex would be peeked into by following the same ordering of the columns of the original matrix. This means that, in the second variation (`depth_charge_1`), DFS would be much more efficient than BFS, taking $1 + d$ steps to find v_{w+d}, while BFS would take $w + d$ steps instead.

I actually don't know whether this is true, as I haven't looked at how *igraph* is implemented. Let's find out empirically, and assess its implications!

Programming hints:

- The solution to Exercise 8 in Exercise Sheet #1 and several exercises in Exercise Sheet #2 (Exercise 2 in particular) provide some ideas on how to run simulations, including a demonstration of `np.random.choice` to simulate drawing integer values (useful to draw whether an edge exists, and which vertex among $v_1, v_2, \ldots, v_w$ will grow a chain). In R, function `sample` is the one to look for.
- The *Graph generation* link above should tell you how to create a *igraph* graph out of an adjacency matrix. For the R version, it should be reasonably clear what the corresponding R function is after you realise what needs to be done in Python.
- to do a z-test in Python, check out this function. For a scatterplot or qqplot, see here and here.
- in R, a simple implementation of a z-test and a demonstration of scatterplot/qqplot can be found here.

If you haven't done any statistics module before and need help with what a scatterplot or a qqplot means, please feel free to contact me.