The University of Melbourne
School of Computing and Information Systems
COMP10002 Foundations of Algorithms
Semester 1, 2023
Assignment 1
Due: 4pm Friday 5 May 2023
Version 1.1

1 Learning Outcomes

In this assignment, you will demonstrate your understanding of arrays, pointers, input processing, and functions. You will also extend your skills in terms of code reading, program design, testing, and debugging.

2 The Story...

Given an array of n sorted numbers, binary search offers a fast $O(\log_2 n)$-time search to look up for a *search key* (that is, a number to be located in the array). This means, given an array with one billion numbers, it takes only $\log_2 10^9 \approx 30$ comparisons to locate a search key. While this is very fast, the modern "big data era" calls for even faster solutions. Think about Black Friday shopping events, Amazon needs to support queries to their inventory with billions of products given a product ID, for millions of users (if not more) at the same time. Google Maps is another example, where millions of users may be querying points of interest given coordinate ranges (e.g., for restaurants within a user's Google Maps app view). Tremendous efforts have been made to scale up these services, including investments on more hardware which later led to the Amazon Web Services.

In this assignment, we will design an algorithmic solution for the problem. Our aim is to further bring down the search time complexity to $O(1)$ (in an ideal case), with a technique called the *learned index*. You do not need to worry about what a learned index is for now. The assignment will guide you through building such an index step by step.

3 Your Task

You will be given a skeleton code file named `program.c` for this assignment on Canvas. The skeleton code file contains a `main` function that has been partially completed. There are a few other functions which are incomplete. You need to add code to all the functions including the `main` function for the following tasks.

The given input to the program consists of 12 lines of *positive* integers as follows:

- The first 10 lines contains 10 unsorted integers separated by whitespace in each line. Together this forms an array of 100 numbers (between 1 and 999, inclusive) to run our search algorithm upon.
- The next line contains a single integer representing a *target maximum prediction error* err_m that we aim to achieve. This integer will be greater than 1. You do *not* need this number until Stage 3.
- The final line contains one or more integers (separated by whitespace) representing the search keys. There is no predefined upper limit on the number of search keys. You do *not* need these search keys until Stage 4.

You may assume that the test data always follows the format as described above. No input validity checking is needed. See the next page for a sample input.

```
164 694 887 133  18 988 851 961 154 223
794 619 973 681 683  93 468 433 873 423
389 465 875 346 347 409  58 374 286 558
607 704 735 631 768 921 247  44 154 464
155 517 551 995 950 132 540 971  64 378
660 164 592 882 594 816 799 685 615   5
 52 691 769 749 297 503 195 785 121 834
356  12 985 975 954 784 800 327 222 735
807 420  98 109 810 934 975 304 282 441
372 970 736  98 685 179 655 500 210 480
5
18 195 735 975 668 1
```

3.1 Stage 1: Read Input Numbers, Sort Them, and Output the First Ten Numbers (Up to 5 Marks)

Your first task is to understand the skeleton code. Note the use of the `data_t` type for generalisability in the skeleton code, which is essentially the `int` type.

Then, you should add code the `stage_one` function to (1) read the first 10 input lines into the `dataset` array, (2) call the `quick_sort` function provided in the skeleton code to sort the `dataset` array in ascending order (*for marking purposes, you are* not *allowed to use sorting code from other sources, or to create your own sorting function from scratch*), and (3) output the first 10 elements in the sorted array.

The sorting step in this stage will help us achieve a fast search over the `dataset` array in the later stages. The output for this stage given the above sample input should be (where "`mac:`" is the command prompt):

```
mac: ./program < test0.txt
Stage 1
==========
First 10 numbers: 5 12 18 44 52 58 64 93 98 98
```

As this example illustrates, the best way to get data into your program is to edit it in a text file (with a "`.txt`" extension, any text editor can do this), and then execute your program from the command line, feeding the data in via input redirection (using `<`). In the program, we will still use the standard input functions such as `scanf` to read the data fed in from the text file. Our auto-testing system will feed input data into your submissions in this way as well. You do not need to (and *should not*) use any file operation functions such as `fopen` or `fread`. To simplify the assessment, your program should not print anything except for the data requested to be output (as shown in the output example).

You should plan carefully, rather than just leaping in and starting to edit the skeleton code. Then, before moving through the rest of the stages, you should test your program thoroughly to ensure its correctness.

You can (and should) create sub-functions to complete the tasks.

3.2 Stage 2: Index with a Single Linear Function (Up to 10 Marks)

As mentioned above, the goal of number searching is to locate a search key in an array of numbers. This translates to returning the subscript index of a number in the array that equals to the search key (if the search key if found). For example, consider the first 10 elements in the sorted `dataset` array from Stage 1, that is, `{5, 12, 18, 44, 52, 58, 64, 93, 98, 98}`, and a search key `key = 18`, a search algorithm should return `2` (the subscript index of `18`), as `dataset[2] = 18`.

A search algorithm thus can be thought of as a *mapping function* $f(key)$ that takes a search key as the input and outputs the subscript index of `key` in array `dataset`. Our core idea to achieve an $O(1)$-time search algorithm is to find a mapping function that runs in $O(1)$-time.

Figure 1 plots a mapping from the first 10 elements of the sorted array `dataset` to their corresponding subscript indices, which forms a (blue solid) polyline. This polyline can be approximated by a single (red dashed) line defined using the two points corresponding to the first two elements of `dataset`, that is, `(0, 5)`

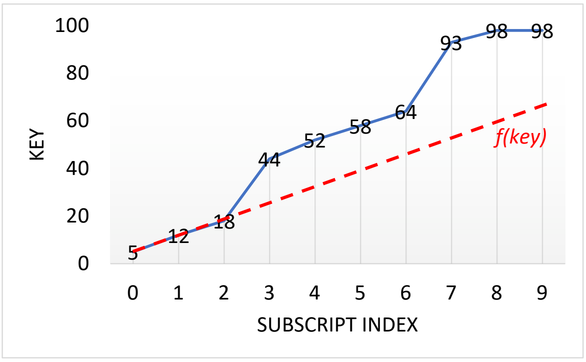

Figure 1: A mapping function example

and `(1, 12)`. Here, `0` and `1` are the subscript indices corresponding to the first two array elements `5` and `12`, that is, `dataset[0] = 5` and `dataset[1] = 12`. The (red dashed) line corresponds to a linear function:

$$f(key) = \frac{(1-0)key + dataset[1] \times 0 - dataset[0] \times 1}{dataset[1] - dataset[0]} = \frac{key - 5}{7} \tag{1}$$

Given a search key `key = 18`, $\lceil f(key) \rceil = 2$ ($\lceil x \rceil$ here is the ceiling function that returns the smallest integer greater than or equal to x), which is the subscript index of `key = 18` in the array – this is an $O(1)$-time search!

Stage 2.1

Equation 1 can be generalised to the form of $f(key) = \dfrac{key + a}{b}$, where a and b are parameters calculated based on the values of `dataset[0]`, `dataset[1]` and their corresponding subscript indices `0` and `1`. For example, `a = -5` and `b = 7` given the sample input.

Write a function that calculates the values of a and b, and a second function that implements the calculation process of $f(key)$, taking key, a, and b as its input parameters.

Note a special case where `dataset[0] = dataset[1]`. To record this special case, you should set $b = 0$ and a to be the subscript index of the first element (that is, `0`), while $f(key)$ should just return a, such that your function f can be generalised in Stage 3.

Stage 2.2

Function $\lceil f(key) \rceil$ may not always return exactly the subscript index of `key` in `dataset`. Continuing with the example above, when `key = 44`, $\lceil f(key) \rceil = 6$, which is not the subscript index of `key`. In this case, we fall back to a scan (or binary search) on both sides of `dataset[6]` until `key` is found or the array boundary has been reached. The data range to be examined is derived as follows.

We call $|\lceil f(key) \rceil -$ `x`$|$ where `dataset[x] = key` the *prediction error* of function f for `key`. In the example above, when `key = dataset[3] = 44`, the prediction error is `|6 - 3| = 3`. We can calculate the prediction error for every number in `dataset`, and record the maximum of all the prediction error values calculated. This maximum error is called the *maximum prediction error* of f, denoted by err_f. At search time, we only need to search within the range of $[\lceil f(key) \rceil - err_f, \lceil f(key) \rceil + err_f]$. In this case, the search time becomes $O(err_f)$ (or $O(\log_2 err_f)$ with binary search, typically $err_f \ll$ `n`).

Now add code to the `stage_two` function to:

1. Compute the maximum prediction error err_f of the function f implemented in Stage 2.1 over the sorted array `dataset`. Hint: You may need the `ceil` and `abs` functions from `math.h` for this computation.
2. Output the computed error err_f, the `dataset` array element with this error, and the subscript index of the element. If there are multiple elements with this error, output the smallest among them.

The output for this stage given the above sample input should be:

```
Stage 2
==========
Maximum prediction error: 46
For key: 950
At position: 89
```

3.3 Stage 3: Index with More Linear Functions to Reduce the Maximum Prediction Error (Up to 17 Marks)

The value of err_f plays a critical role in the search efficiency, and different strategies have been proposed to limit this value. In this stage, you will implement a strategy that uses different mapping functions for different segments of the `dataset` array, such that the maximum prediction error of each mapping function is bounded by a given *target maximum prediction error* err_m.

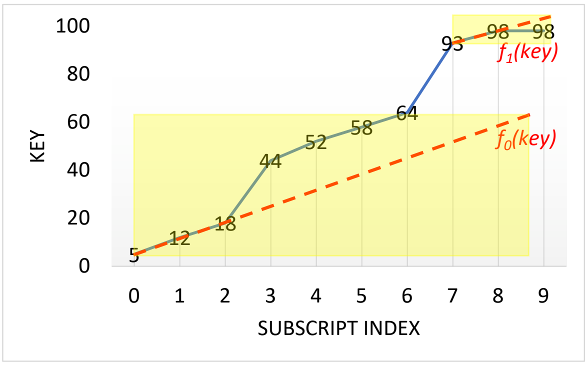

Figure 2: Indexing with multiple mapping functions

Add code to the `stage_three` function to read err_m given in the input and implement a strategy that achieves this target as follows.

1. Compute the values of parameters a and b for a function $f(key) = \dfrac{key + a}{b}$ using the first two elements of the `dataset` array as done in Stage 2.

2. Starting from the third element of the `dataset` array, compute the prediction error of f for the element. If the prediction error does not exceed err_m, we say that the element is "*covered*" by function f. Otherwise, we say that the element *cannot* be covered by f.

3. Repeat Step 2 with the rest of the elements until the first element that cannot be covered by f is found. Let this element be `dataset[i]`. Continuing with the sample input, for `dataset[6] = 64`, $|\lceil f(64) \rceil - 6| = |\lceil \frac{64-5}{7} \rceil - 6 = 3 \leq err_m = 5.$, while for `dataset[7] = 93`, $|\lceil f(93) \rceil - 7| = |\lceil \frac{93-5}{7} \rceil - 7 = 6 > err_m = 5$. Thus, the first element that cannot be covered by f is `dataset[7]`, i.e., `i = 7`.

4. Store the parameter values of a and b as well as the value of `dataset[i-1]` calculated so far. These values define the first segment of `dataset` covered by our first mapping function, as shown by $f_0(key)$ in Figure 2. We will need to store more triples of `(a, b, dataset[i-1])` for the later segments in the next steps. To store all such triples, a `struct` typed array will do this nicely. Read Chapter 8 of the textbook if you would like to take this approach, or you can use three arrays for the same purpose. You will need to work out a proper size for the array(s). *Hint:* You can work out this size based on the input data format as described earlier. You do *not* dynamic memory allocation for this assignment.

5. Restart from Step 1, this time using `dataset[i]` and `dataset[i+1]` to compute a new mapping function. With the example shown in Figure 2, this means computing the parameter values a and b for function $f_1(key) = \dfrac{key + a}{b}$, using `dataset[7]`, `dataset[8]`, and their subscript indices 7 and 8. You need to analyse Equation 1 to generalise the computation of a and b.

   Don't forget about the special case where `dataset[i] = dataset[i+1]`. Following Stage 2.1, in this case, we should set $b = $ `0` and $a = $ `i`.

   If there is just one element left, we should set $b = $ `0` and $a = $ `n - 1`. Recall that `n` is the number of data elements in the `dataset` array.

6. The process above continues, and multiple mapping functions will be created, each recorded in the formed of a triple `(a, b, dataset[i-1])` as mentioned above. The algorithm should terminate when no more points are to be covered by new mapping functions.

For this stage, the output of your code should be the target maximum prediction error err_m and the list of mapping functions computed. For example, given the sample input above, the output of this stage is shown in the next page.

```
Stage 3
==========
Target maximum prediction error: 5
Function  0: a =   -5, b =   7, max element =  64
Function  1: a =  -58, b =   5, max element = 133
Function  2: a =   14, b =   0, max element = 179
Function  3: a =  105, b =  15, max element = 468
Function  4: a =  420, b =  20, max element = 683
Function  5: a =   62, b =   0, max element = 735
Function  6: a = -667, b =   1, max element = 736
Function  7: a =  581, b =  19, max element = 810
Function  8: a =  624, b =  18, max element = 973
Function  9: a =   95, b =   0, max element = 995
```

Hint: For debugging purposes, you may also print out the full sorted `dataset` array. Make sure to remove the extra `printf` statements before making your final submission.

3.4 Stage 4: Perform Exact-Match Queries (Up to 20 Marks)

We have constructed our index structure above. In this stage, we implement a search algorithm with the structure, by adding code to the `stage_four` function.

Your `stage_four` function should read each integer (that is, a search key) from the last line of the input. For each search key `key`, the search process runs as follows:

1. If `key` is smaller than the minimum element or greater than the maximum element in `dataset`, output "`not found!`" and terminate the search.
2. Otherwise, run a binary search over the array of mapping functions (using the maximum `dataset` array element covered by each function) to locate the mapping function f covering `key`. Continuing with our example, this means to run a binary search over the array of "`max element`"s `{64, 133, 179, 468, 683, 735, 736, 810. 973, 995}` produced by Stage 3.
3. Run a binary search over $[\max\{0, \lceil f(\texttt{key}) \rceil - err_m\}, \min\{\texttt{n}-1, \lceil f(\texttt{key}) \rceil + err_m\}]$ to locate `key` from the `dataset` array. Here, max and min are functions to calculate the maximum and minimum value between two numbers, respectively. They help guard the search range to be within the array boundary `[0, n-1]`. You need to write code to implement these functions.

Note: You should adapt the `binary_search` function included in the skeleton code for the two steps above, to output the `dataset` array elements that have been compared with during the search process. You can create two binary search functions for the two steps separately (without penalties on duplicate code). *For marking purposes, you are* not *allowed to use binary search code from other sources, or to create your own binary search functions from scratch.*

The output for this stage given the sample input above is shown in the next page.

Take searching for `key = 18` as an example. At Step 2, we start by comparing with the maximum `dataset` array element covered by mapping function `(0 + 10)/2 = 5` (given that there are 10 mapping functions), which is `735 > 18` (see Stage 3 output). We next consider mapping function `(0 + 5)/2 = 2`, which covers `dataset` array elements up to `179 > 18`. Continuing with this process, mapping function `(0 + 2)/2 = 1` covering up to `133 > 18` is the next to considered, followed by mapping function `(0 + 1)/2 = 0` covering up to `64 > 18`. After this, our search range will become `[0, 0]`, which means that there is just mapping function `0` to be considered. This is the mapping function covering `key = 18`, and $\lceil$`f(18)`$\rceil$ `= 2` (recall the calculation in Stage 2).

At Step 3, a binary search is run over `dataset` in the range of $[\max\{0, \lceil f(\texttt{key}) \rceil - err_m\}, \min\{\texttt{n}-1, \lceil f(\texttt{key}) \rceil + err_m\}] = [0, 7]$. The first array element to compare with is `dataset[(0 + 8)/2] = 52 > 18`. Next, we compare with `dataset[(0 + 4)/2] = 18`, and the search key is found.

```
Stage 4
==========
Searching for 18:
Step 1: search key in data domain.
Step 2: 735 179 133 64
Step 3: 52 18 @ dataset[2]!
Searching for 195:
Step 1: search key in data domain.
Step 2: 735 179 683 468
Step 3: 195 @ dataset[20]!
Searching for 735:
Step 1: search key in data domain.
Step 2: 735
Step 3: 685 694 735 @ dataset[67]!
Searching for 975:
Step 1: search key in data domain.
Step 2: 735 973 995
Step 3: 975 @ dataset[95]!
Searching for 668:
Step 1: search key in data domain.
Step 2: 735 179 683 468
Step 3: 615 655 681 660 not found!
Searching for 1:
Step 1: not found!
```

What you have implemented above is the core idea of the so-called *learned indices*, which is a latest development of data indexing techniques based on machine learning.[1] One of your lecturers, Dr. Jianzhong Qi, contributed to this area of research by introducing learned indices into multidimensional data indexing. See `https://people.eng.unimelb.edu.au/jianzhongq/papers/ICDE2023_ELSI.pdf` for his latest work on how to construct learn indices for multidimensional data efficiently.

Open challenge (no answer needed in your assignment submission, but you are welcomed to share your thoughts on Ed with *private* posts): As you may have observed, using our structure may end up with close to $\lceil \log_2 100 \rceil = 7$ comparisons at times which means it is not always better than a naive binary search. This is because we have simplified the learned index structure for the assignment purpose. In practice, we can choose the array elements to compute a mapping function with smarter strategies, such that each mapping function can cover more array elements, and hence there are fewer mapping functions to examine for Step 2 above. Also, the maximum elements of the mapping functions form another array, which can be indexed by another layer of mapping functions, and we can do this recursively to form a hierarchical structure. Can you think of other strategies to make learned indices even better?

4 Submission and Assessment

This assignment is worth 20% of the final mark. A detailed marking scheme will be provided on Canvas.

Submitting your code. To submit your code, you will need to: (1) Log in to Canvas LMS subject site, (2) Navigate to “Assignment 1” in the “Assignments” page, (3) Click on “Load Assignment 1 in a new window”, and (4) follow the instructions on the Gradescope “Assignment 1” page and click on the “Submit” link to make a submission. You can submit as many times as you want to. *Only the last submission made before the deadline will be marked.* Submissions made after the deadline will be marked with late penalties as detailed at the end of this document. Do *not* submit after the deadline unless a late submission is intended. Two hidden tests will be run for marking purposes. Results of these tests will be released after the marking is done.

You can (and should) submit both **early and often** – to check that your program compiles correctly on our test system, which may have some different characteristics to your own machines.

[1]Machine learning in (perhaps overly) simple terms is just to fit the parameter values of mathematical models (e.g., a linear mapping function) to a given set of data samples a.k.a. training data (e.g., an array of numbers).

Testing on your own computer. You will be given a sample test file `test0.txt` and the sample output `test0-output.txt`. You can test your code on your own machine with the following command and compare the output with `test0-output.txt`:

```
mac: ./program < test0.txt    /* Here '<' feeds the data from test0.txt into program */
```

Note that we are using the following command to compile your code on the submission testing system (we name the source code file `program.c`).

```
gcc -Wall -std=c17 -o program program.c -lm
```

The flag "`-std=c17`" enables the compiler to use a modern standard of the `C` language – `C17`. To ensure that your submission works properly on the submission system, you should use this command to compile your code on your local machine as well.

You may discuss your work with others, but what gets typed into your program must be individual work, **not** from anyone else. Do **not** give (hard or soft) copies of your work to anyone else; do **not** "lend" your memory stick to others; and do **not** ask others to give you their programs "just so that I can take a look and get some ideas, I won't copy, honest". The best way to help your friends in this regard is to say a very firm "no" when they ask for a copy of, or to see, your program, pointing out that your "no", and their acceptance of that decision, is the only thing that will preserve your friendship. *A sophisticated program that undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions in "compare every pair" mode.* See `https://academichonesty.unimelb.edu.au` for more information.

Deadline: Programs not submitted by **4pm Friday 5 May 2023** will lose penalty marks at the rate of 3 marks per day or part day late. Late submissions after 4pm Monday 8 May 2023 will **not** be accepted. Students seeking extensions for medical or other "outside my control" reasons should email the lecturer at jianzhong.qi@unimelb.edu.au. If you attend a GP or other health care professional as a result of illness, be sure to take a Health Professional Report (HRP) form with you (get it from the Special Consideration section of the Student Portal), you will need this form to be filled out if your illness develops into something that later requires a Special Consideration application to be lodged. You should scan the HPR form and send it in connection with any non-Special Consideration assignment extension requests.

And remember, *Algorithms are fun!*