Assignment 5 - Camellia sinensis

Due: 11:59PM Friday 4th June 2021 Sydney time

This assignment is worth 15% of your final assessment

Task description

In this assignment you will be implementing a simple encrypted key-value store. Your code will be a library of functions that can be called by other programs, rather than a standalone executable. Your library will also be assessed for performance and thread safety. Concurrent requests may be made to your dictionary, and you may also wish to implement concurrency within your library itself.

You will already be familiar with the dictionary abstract data type which maps unique keys to corresponding values, and supports insertion, deletion and searching. You will implement your dictionary using the B tree data structure and encrypt stored values using a modified version of the Tiny Encryption Algorithm (TEA).

B trees

The B tree is a self-balancing tree that stores key-value pairs in its nodes. When we refer to “keys” below, we implicitly also mean the value that each key is linked to. Comparisons on keys only operate on keys, not their linked values.

Each node may have a varying number of children, which is dictated by the branching factor `b`, a positive integer such that $b \geq 2$. `b` is a fixed property of the entire tree at creation. Each node obeys the following rules:

- The value `n` of every node obeys $\lceil b/2 \rceil \leq n \leq b$ and for internal nodes, `n` is the number of children they have.
- The exception is the root node. If it is a leaf, it obeys $n \leq b$. If it is not a leaf, it obeys $2 \leq n \leq b$.
- Every node has `n - 1` keys. This rule also applies to leaf nodes; they do not have `n` children, but they still contain `n - 1` keys where `n` satisfies the rules above.

Every key exists in exactly one node in the tree. The keys in each node are ordered and for internal nodes, partition their children into an ordering. Suppose the n - 1 keys of an internal node satisfy the order $k_0 < k_1 < \ldots < k_{n-2}$, and suppose the n children are C_0, $C_1 \ldots C_{n-1}$. Every key in a child node C_i is $> k_{i-1}$ and $< k_i$. Every key in the first child (C_0) is $< k_0$ and every key in the last child (C_{n-1}) is $> k_{n-2}$. We will use the terminology that the key k_i separates the children C_i and C_{i+1} and that the child C_i is separated by the keys k_{i-1} and k_i (or only one key if it is the first or last child).

Searching a B tree

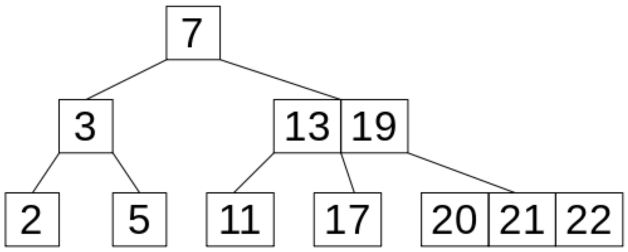
To locate a key K in a B tree, follow this algorithm:

1. Starting from the root node, if the key is present in this node, the key-value pair is returned and we are done. If it is not present, identify the correct child to consider next by comparing K with the node’s keys. If $k_{i-1} < K < k_i$, then the correct child is C_i.
2. Move to C_i, and repeat step 1 until the node containing K is found.
3. If a leaf node is reached and K is not found, return key not found.

Example of B tree and searching

Below is a B tree with branching factor 4. Keys are presented ordered, note how the keys in each node separate the children according to the keys in the children (separators indicated by the source of each edge at the parent). Each key is linked to its value; this is not shown in the diagram.

To search for the value 17 in this tree, start from the root. 17 > 7, so move to the child with keys 13 and 19. 13 < 17 < 19, so move to the second child. Here we find the key 17.



Inserting into a B tree

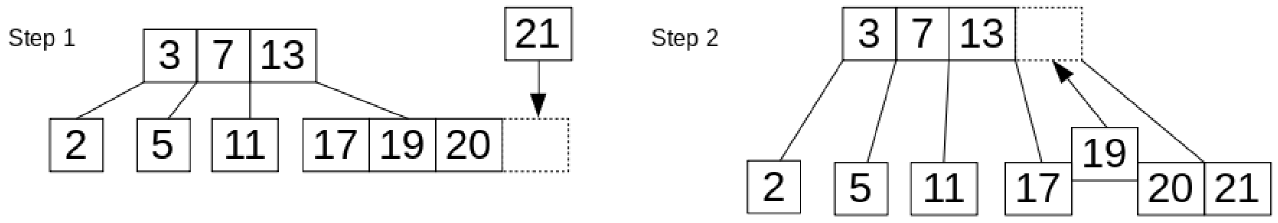
To insert a key K with value V into a B tree, follow this algorithm:

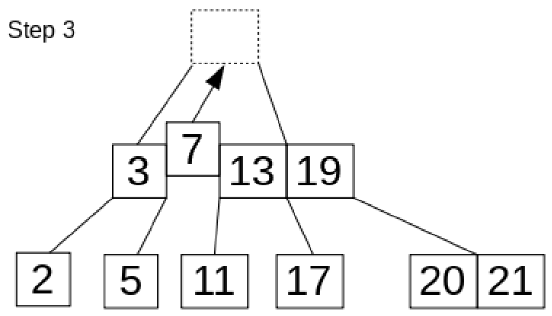
1. First, follow the searching algorithm to search for K in the tree. It is an error if K already exists in the tree. Identify the leaf node that would contain K.
2. Insert K and V into the node, maintaining the ordering of keys
3. If the new number of keys in the node does not exceed the limit b - 1, the insertion algorithm is complete
4. If the number of keys exceeds the limit b - 1, identify the median of the keys including the new key, K_{median}. If the total number of keys is even, choose the smaller of the two middle keys as K_{median}. Split the node into two new sibling nodes of the same parent with the new left node containing all keys < K_{median} and the new right node containing all keys > K_{median}. Move K_{median} into the parent node's keys. Note that K_{median} now separates the two new nodes.
5. If the parent node's number of keys also exceeds the limit, then repeat step 4 recursively up the tree. As internal nodes are split, their children are also relinked as appropriate to the new sibling nodes.
6. If the recursive splitting reaches the root node, it is split into two siblings, and the K_{median} value is inserted into a new root as the only key. The new root has the two new nodes as its only children. The height of the entire tree therefore increases by 1.

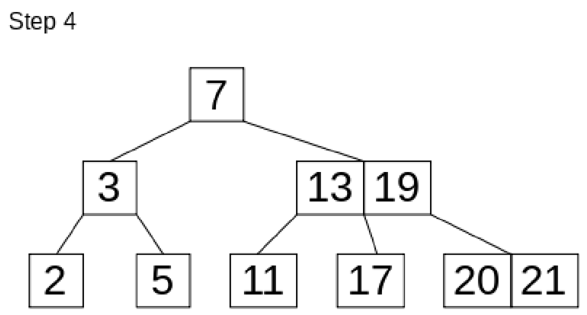
Example of insertion into a B tree

We want to insert the key 21 into the B tree below with branching factor 4. A search for 21 reveals the target node for insertion is the one containing the keys 17, 19, 20. Insertion of 21 would cause this node to overflow. K_{median} is 19 (note the even number of keys, so choose the smaller of the 2 middle keys). 19 is moved to the parent node, and the remaining keys are split into two siblings (node with 17, and node with 20, 21).

The key 19 is moved to the parent, the root, which also overflows and needs to split. The K_{median}, 7, is moved up to become a new root, separating the new siblings (node with 3, and node with 13, 19).







Deletion from a B tree

To delete a key K from a B tree, follow this algorithm:

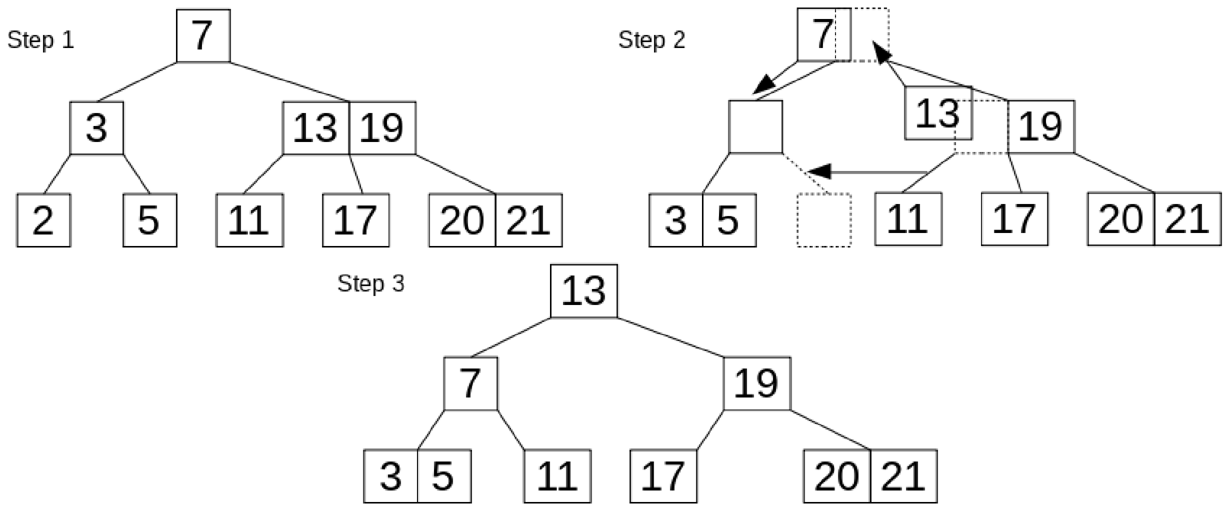
1. First, follow the searching algorithm to search for K in the tree. It is an error if K does not exist in the tree.
2. If the node containing K is an internal node, suppose that C is the left of the 2 child nodes that K separates. Swap K with the largest key (K_{new}) in the entire subtree rooted at C. Note that K_{new} must reside in a leaf node, so after the swap K now resides in this leaf node. Note that now K_{new} also correctly separates the same 2 child nodes as K did.
3. If K was in an internal node, note that K is now in a leaf node and this is identical to the case where K was originally in a leaf node. Delete K from the leaf node. If the leaf node still satisfies the minimum number of keys, the deletion algorithm is complete.
4. After deletion, the leaf node (call this the target node) may now have less than the minimum number of keys permitted. Suppose that the target node is separated by the keys K_{left} and K_{right} in the parent. Firstly, check if the immediate left sibling of the target node (by order) has more than the minimum number of keys. If it does, in the parent replace K_{left} with the largest key in the left sibling (call this $K_{leftchild}$), and move K_{left} into the target node so that it has the minimum number of keys required. If the immediate left sibling is unsuitable, if the immediate right sibling has more than the minimum number of keys, in the parent replace K_{right} with the smallest key in the right sibling (call this $K_{rightchild}$), and move K_{right} into the target node. If the target node is the leftmost or rightmost child of the parent, skip the check of the immediate left or immediate right siblings respectively. If a suitable sibling was identified and the keys moved according to this step, the deletion algorithm is complete. If a suitable sibling is not identified, continue to step 6.
5. For future recursive steps, step 4 may need to be performed on a target node that is internal as follows. If the left sibling has more than the minimum number of keys, let C_{left} be its "largest" child, which is separated in the left sibling by $K_{leftchild}$. Note that, due to the insertion algorithm, all keys in C_{left} are also < K_{left}. Then, according to step 4, K_{left} is moved into the target node, and $K_{leftchild}$ is moved into the parent. If the target node is internal, also move the child (and subtree rooted at) C_{left} so that it is now the leftmost child of the target node separated by the new key K_{left}. Likewise if we are considering the right sibling, let C_{right} be the "smallest" child of the right sibling, which is separated by $K_{rightchild}$. K_{right} is moved into the target node, $K_{rightchild}$ is moved to the parent, and C_{right} (and its subtree) is moved to become the rightmost child of the target node, separated by K_{right}. Again, if this step completes successfully, the deletion algorithm is complete.
6. If there is no immediate sibling of the target node that has more than the minimum number of keys, merge the target node with an immediate sibling. Always pick the left sibling, unless the target node is the leftmost child of the parent, in which case pick the right sibling to merge with. Move all keys and children from the sibling to the target node. Additionally, move the parent key that separates the target node and the merged sibling, into the target node. That is, if we are merging the target node with its left sibling, then K_{left} must be moved from the parent into the new merged node. Similarly, move K_{right} into the new merged node if required. The parent has now lost a key. If it has less than the minimum number,

repeat steps 4-6 recursively up the tree. If it satisfies the minimum number of keys, the algorithm is complete.

7. If the root node is reached and has less than the minimum number of keys (recall the minimum number of keys for the root specifically is 1), simply delete the root node. The new root is the merged child node produced from step 6. The height of the entire tree therefore decreases by 1.

Example of deletion from a B tree

We want to delete the key 2 from the B tree below with branching factor 4. Removal of the key 2 from its leaf node causes the node to underflow. The only immediate sibling, node containing 5, is also at its minimum number of keys. Therefore by step 6, we merge with the right sibling containing 5 and the key 3 from the parent, as 2 was the leftmost child. The parent node has now underflowed (referred to from here on as the target node). Its right sibling, node containing 13, 19, has more than the minimum number of keys. This is an internal node, so following step 5, we move K_{right} (7) into the target node, $K_{rightchild}$ (13) into the parent of the target node, and C_{right} (containing the node with key 11) to become the rightmost child of the target node. The target node now has key 7 which separates node containing 3, 5 and node containing 11.



Tiny Encryption Algorithm (modified)

Encryption is simply a function ("cipher") which transforms input data to be encrypted ("plaintext") into encrypted output data ("ciphertext") under the control of some additional data ("key"). The operation can be reversed, which for our purposes takes the ciphertext and applies a decryption function with the same key data, to produce the same original plaintext. TEA is a 64-bit block cipher with 128-bit key size. This means the encryption function always operates on a fixed size block of 64 bits of plaintext and 128 bits of key and outputs 64 bits of ciphertext. The decryption function accepts 64 bits of ciphertext and 128 bits of key and outputs 64 bits of plaintext.

The encryption function has the following pseudocode:

```
// plain contains the 64 bit plaintext
uint32_t plain[2] // Represent 64 bit plaintext as array of 2 uint32_t
uint32_t cipher[2] // Represent 64 bit ciphertext as array of 2 uint32_t
uint32_t key[4] // Represent 128 bit key as array of 4 uint32_t
sum = 0
delta = 0x9E3779B9
cipher[0] = plain[0]
```

```
cipher[1] = plain[1]
loop 1024 times:
    sum = (sum + delta) mod 2^32
    tmp1 = ((cipher[1] << 4) + key[0]) mod 2^32
    tmp2 = (cipher[1] + sum) mod 2^32
    tmp3 = ((cipher[1] >> 5) + key[1]) mod 2^32
    cipher[0] = (cipher[0] + (tmp1 XOR tmp2 XOR tmp3)) mod 2^32
    tmp4 = ((cipher[0] << 4) + key[2]) mod 2^32
    tmp5 = (cipher[0] + sum) mod 2^32
    tmp6 = ((cipher[0] >> 5) + key[3]) mod 2^32
    cipher[1] = (cipher[1] + (tmp4 XOR tmp5 XOR tmp6)) mod 2^32
// cipher now contains the 64 bit ciphertext
```

The decryption function has the following pseudocode:

```
uint32_t cipher[2] // Represent 64 bit ciphertext as array of 2 uint32_t
uint32_t plain[2] // Represent 64 bit plaintext as array of 2 uint32_t
uint32_t key[4] // Represent 128 bit key as array of 4 uint32_t
sum = 0xDDE6E400
delta = 0x9E3779B9
loop 1024 times:
    tmp4 = ((cipher[0] << 4) + key[2]) mod 2^32
    tmp5 = (cipher[0] + sum) mod 2^32
    tmp6 = ((cipher[0] >> 5) + key[3]) mod 2^32
    cipher[1] = (cipher[1] - (tmp4 XOR tmp5 XOR tmp6)) mod 2^32
    tmp1 = ((cipher[1] << 4) + key[0]) mod 2^32
    tmp2 = (cipher[1] + sum) mod 2^32
    tmp3 = ((cipher[1] >> 5) + key[1]) mod 2^32
    cipher[0] = (cipher[0] - (tmp1 XOR tmp2 XOR tmp3)) mod 2^32
    sum = (sum - delta) mod 2^32
plain[0] = cipher[0]
plain[1] = cipher[1]
// plain now contains the 64 bit plaintext
```

Treat all integers in the TEA algorithms above as little endian.

Encrypting arbitrary data with TEA

For this assignment, you will extend the basic TEA algorithm, which operates on 64 bit blocks of data, to encrypt arbitrary sized data. You will do this by using the “counter mode” of block cipher operation (“TEA-CTR”). Suppose that the basic TEA encryption function is `TEA_encrypt(plaintext, key)`, which accepts 64 bit plaintext, 128 bit key, and returns 64 bit ciphertext. The inputs are P (data to be encrypted of an arbitrary number of bytes in size), key (128 bit key), and nonce (64 bits of data which is provided by the caller). First, divide P into 64 bit chunks: P[0], P[1], ... ; the final chunk may contain less than 64 bits of plaintext. For the purposes of the algorithm, fill the remainder of the final chunk with zero bytes.

The output of the algorithm is C, an array of 64 bit chunks C[0], C[1], ... of the same number of chunks as P. Truncate the final chunk of C such that it is of the same length as the original final chunk of P. Therefore, the final C is exactly the same size as the original input P.

The following pseudocode describes TEA-CTR encryption:

```
uint64_t P[number of chunks] // Data to be encrypted as 64 bit chunks
uint32_t key[4] // 128 bit key which is provided
uint64_t nonce // 64 bit "nonce" data which is provided
for i in 0 ... number of chunks:
    tmp1 = i XOR nonce
    tmp2 = TEA_encrypt(tmp1, key)
    C[i] = P[i] XOR tmp2
```

Decryption for TEA-CTR is extremely similar to the encryption algorithm and can be quickly deduced from the above.

Dictionary interface

The keys of your dictionary are unsigned 32-bit integers. Each value corresponding to a key will contain the following elements:

- Size of the actual data for this value in bytes (the maximum size is 2^{32}-1 bytes)
- The encryption key
- The “nonce” value used for TEA-CTR
- The actual encrypted data for this value

These values need to be associated with their key, but how and where exactly you store this data is up to you. The implementation of the main B tree is also up to you, and you may use any other data structures you wish. There are functions you need to implement that will inspect the correctness of your B tree structure.

Functions to implement

Implement the following functions for your library. Do not write any `main()` function. Other programs will directly call your functions.

```
void * init_store(uint16_t branching, uint8_t n_processors);
```

This function will be called exactly once before any other functions are called.

Initialise any data structures and perform any preparation you wish. Return a pointer (`void *`) to a memory area where you store data you wish to use for other functions. For all other functions, this pointer will be passed in for you to use as the `void * helper` parameter. The `branching` parameter gives the branching factor b for your B tree. The `n_processrs` parameter gives you an indication of the number of physical processors you have access to. You have the opportunity to setup any concurrency at this point to use in further operations.

```
void close_store(void * helper);
```

This function will be called exactly once after the end of all other functions. You need to perform any cleanup required of your library, including deallocating dynamic memory.

```
int btree_insert(uint32_t key, void * plaintext, size_t count, uint32_t encryption_key[4],
uint64_t nonce, void * helper);
```

Insert `key` into your B tree with the given `plaintext` data of size `count` bytes. You must encrypt the contents of `plaintext` using a single instance of the TEA-CTR algorithm with the parameters given in `encryption_key` and `nonce` and store the encrypted result in your tree. You cannot store the pointer `plaintext` itself in your tree, it may not remain valid. Return 0 if the insertion is successful. Return 1 if the key already exists. For error cases, the B tree should remain unchanged and ready for further requests.

```
int btree_retrieve(uint32_t key, struct info * found, void * helper);
```

Search for `key` in the B tree and fill its value into the `struct info` pointed to by `found`. For the `data` field, return a pointer to the encrypted data, do not decrypt it. If successful, return 0, otherwise, return 1. For error cases, the B tree should remain unchanged and ready for further requests. `struct info` is the following:

```
struct info {
    uint32_t size; // Size of actual stored data in bytes
    uint32_t key[4]; // Encryption key
    uint64_t nonce; // Nonce data
    void * data; // Pointer to the encrypted stored data
};
```

```
int btree_decrypt(uint32_t key, void * output, void * helper);
```

Search for `key` in the B tree and decrypt its data, copying the plaintext to `output`, which you can assume has sufficient space for the size of the data stored for this key. The data in the tree must remain encrypted. If successful, return 0, otherwise return 1. For error cases the B tree should remain unchanged and ready for further requests.

```
int btree_delete(uint32_t key, void * helper);
```

Delete the `key` and its value from the B tree. If successful, return 0, otherwise return 1. For error cases the B tree should remain unchanged and ready for further requests.

```
uint64_t btree_export(void * helper, struct node ** list);
```

Export the B tree as an array of `struct node` (described below). The array must contain the nodes in your B tree, ordered as a preorder traversal. Each node in your tree corresponds to a `struct node` which contains the number of keys it has and an ordered array of the keys. Your function must cause `*list` to point to the first element of your array. Your array must be dynamically allocated; the caller will call `free()` on your array. The `keys` array of each `struct node` should also be dynamically allocated; the caller will call `free()` on each of these pointers. The return value must be the number of nodes in your B tree. If the B tree has no nodes, then `*list` can be NULL and `free()` will not be called.

```
struct node {
    uint16_t num_keys; // The number of keys in this node
    uint32_t * keys; // The keys in this node, ordered
};
```

An example of how `btree_export` would be tested follows:

```
// This is testing code
struct node * list = NULL;
uint64_t num = btree_export(helper, &list);
// num and elements of list e.g. list[i] will be inspected
free(list); // expected that list will point to dynamic memory
```

```
void encrypt_tea(uint32_t plain[2], uint32_t cipher[2], uint32_t key[4]);
```

Using TEA, encrypt the 64 bit plaintext provided in `plain`, using the 128 bit key provided in `key`. Write the 64 bit ciphertext output to `cipher`.

```
void decrypt_tea(uint32_t cipher[2], uint32_t plain[2], uint32_t key[4]);
```

Using TEA, decrypt the 64 bit ciphertext provided in `cipher`, using the 128 bit key provided in `key`. Write the 64 bit plaintext output to `plain`.

```
void encrypt_tea_ctr(uint64_t * plain, uint32_t key[4], uint64_t nonce, uint64_t * cipher,
uint32_t num_blocks);
```

`num_blocks` blocks of 64 bit input plaintext are provided in `plain`. Using TEA-CTR, encrypt this input using the 128 bit key provided in `key` and the 64 bit `nonce`. Write the output ciphertext blocks to `cipher` which you can assume has enough space.

```
void decrypt_tea_ctr(uint64_t * cipher, uint32_t key[4], uint64_t nonce, uint64_t * plain,
uint32_t num_blocks);
```

`num_blocks` blocks of 64 bit input ciphertext are provided in `cipher`. Using TEA-CTR, decrypt this input using the 128 bit key provided in `key` and the 64 bit `nonce`. Write the output plaintext blocks to `plain` which you can assume has enough space.

Note on synchronisation

The tests performed on your library functions may be multithreaded; after initialisation, any functions may be called concurrently. For example, an insertion may be requested while a retrieval is proceeding in another testing thread. All your library functions are expected to behave atomically. You must implement synchronisation primitives to ensure concurrent requests are handled correctly and not interleaved. It is the caller's responsibility to ensure that they order their calls correctly to achieve the desired result, but it is your responsibility to ensure that concurrent operations behave sequentially and atomically. For performance reasons, you may also wish to multithread internally within your own functions. This is separate from your code being tested concurrently.

Test Cases

You must write your own test cases for this assignment. Details on execution and marking of your test cases are included below.

Code Submission and Marking Criteria

Only the last submission will be graded. Late submissions will incur the late penalty described in the Unit of Study outline. Submit your assignment on Ed via git. It must compile and run on the Ed submission system. For Ed testing, your code will be compiled to a static library (`libbtreestore.a`) with the Makefile rule `make correctness`; a scaffold is provided.

For Ed testing, your code will compile with these options:

```
-O0 -Werror=vla -std=gnu11 -g -fsanitize=address -pthread -lrt -lm
```

Performance testing will also be performed, for which your code will compile with the Makefile rule `make performance`, with these options:

```
-O0 -march=native -Werror=vla -std=gnu11 -pthread -lrt -lm
```

Your library must declare its functions in the header `btreestore.h` (provided in scaffold). The scaffold contains tests.c which includes sample main() code that calls your library functions, which you can optionally use for your testing. For testing purposes, `make tests` will compile your library. Your test cases are expected to link your library for testing. Test cases you write must be executed by `make run_tests`, which should report back on their results in a human readable format. You can modify your Makefile, but you cannot change the compiler or remove compilation flags for `make correctness` or `make tests`. You cannot change any compilation flags for `make performance`. Remember that all testing will be performed against the static library `libbtreestore.a`.

You may not use variable length arrays or alloca.

Failing to adhere to these rules may cause you to receive 0.

Marking criteria follow (15 marks total):

- 8 marks for correctness (passing Ed automatic test cases). Some test cases will be hidden and not available before or after the deadline. The total number of tests and the proportion of public versus hidden tests are not specified.
- 5 marks for performance. You are only eligible for these marks if you pass all the correctness test cases. Submissions will be compared against defined cutoffs, with increasing amounts of credit available for better speed of execution against a range of test cases.
  - Your code will be assessed for performance by recompiling and running on a physical computer with an Intel Core i7-8700 CPU and 16GB of RAM with minimal other processes running. The operating system will be a Debian Linux 10.9 installation, with the following system software versions: Linux kernel 4.19, GNU C library 2.28, GCC 8.3. More details about performance marking will be provided at a later date.
- 2 marks for your own automated test cases, with the library built with `make tests` and your test cases built and/or executed with `make run_tests`. The proportion of marks for this section will be capped to the proportion of marks that you achieve out of 8 for the correctness portion.
- There is no oral session

Warning: Any attempts to deceive or disrupt the marking system will result in an immediate zero for the entire assignment. Negative marks can be assigned if you do not follow the assignment problem description or if your code is unnecessarily or deliberately obfuscated.

Hints

- You are allowed to use all features of the specified CPU, including SIMD instructions. Please note for correctness your code must still compile and run on Ed.

Academic Declaration

By submitting this assignment you declare the following:

I declare that I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure, and except where specifically acknowledged, the work contained in this assignment/project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgment from other sources, including published works, the Internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.