TÉCNICO LISBOA

DEI
Departamento de Engenharia Informática

Advanced Algorithms

2022/2023

Project 3

Circular String Linearization

This project considers the problem of finding a canonical representation for circular string in linear time. The first delivery deadline for the project code is April 14, at 12:00. The deadline for the recovery season is 14 July at 12:00, the system opens the 12th July at 12:00. The deadline for the special season is 26 July at 12:00, the system opens the 24th July at 12:00.

Students should not use existing code for the algorithms described in the project, either from software libraries or other electronic sources. They should also not share their implementation with colleagues, specially not before the special season deadline.

It is important to read the full description of the project before starting to design and implementing solutions.

The delivery of the project is done in the mooshak system.

1 Suffix Trees

In this problem we are given some linearization of a string and seek to find its smallest lexicographic cyclic rotation. To solve this problem we will need to implement the suffix tree data structure in linear time. An algorithm for this problem is explained in section 7.6 of Gusfield's book [1], but the implementation must closely follow the description given in this script. This description gives a guided implementation of Ukkonen's algorithm. Implementations that use variations of the data structures we proposed usually lead to hard to implement and debug solutions, thus we advise following the

given description very closely.

The input will consist of a set of DNA sequences, hence the underlying alphabet will be A,C,T,G.

In this project we do not require a naive implementation, but students are advised to implement one for debugging purposes. In a naive implementation the tree is built by inserting one suffix at a time into a trie.

1.1 Description

The input contains a text string `T`, we want to produce an explicit suffix tree for the string `T.T`, i.e., the string `T` concatenated with itself. The size of the input string is `n`.

Ukkonen's algorithm is fairly challenging to implement and constitutes the major part of this assignment. In implementing this data structure students are advised to use the "sentinel" programming technique. This technique consists in augmenting the data structures to avoid writing more code. The idea is that to avoid producing code that contains many conditional selections, `if` statements mainly, we add extra content in the data structure. An important example for suffix trees is the suffix link of the `root` node. The general definition does not apply to this case, therefore it is formally undefined. For programming purposes it is best to add a `sentinel`, that is the suffix link of the `root`. It should also be possible to `Descend` from this node to the `root` with any letter.

A fair amount of information must be stored at each node. A recommended structure, in C, is the following:

```
typedef struct node* node;

struct node
{
  int head; /**< The path-label start at &(Ti[head]) */
  int sdep; /**< String-Depth, from the root down to node */
  node child; /**< First child */
  node brother; /**< Brother */
  node slink; /**< Suffix-Link */
  node* hook; /**< What keeps this node linked? */
};
```

Most fields are explained by the comments. This structure can be used both to represent internal nodes and to represent leaves. For leaves a smaller structure could be used, thus saving overall space, but such optimization is not critical and therefore not recommended. The `head` field indicating the

position on `T` where the path-label, from the root down to the node, starts. Recall that in `C` arrays start at position 0. The `sdep` field stores the string depth of the node, i.e., the number of letters that exist by start counting at the root until we reach the node. For the `root` node the string-depth is 0, which represents the fact that the path-label is empty. The `child` and `brother` pointers are used to implement a singly linked list. The `child` points to the first element of the list and the `brother` to the next element on the list. This list contains the nodes that one can access when leaving the current node. This list is scanned when we are searching to descend by a particular letter. In general the order of the list is not structured which means that it will be necessary to scan the whole list when search for a letter that is not present. The trickiest field in the previous structure is the `hook`, which is a pointer to a pointer to a `struct`. A node `struct` is pointed to by only one `child` or `brother` pointer, the idea of the `hook` is to keep a pointer to that pointer. With this information it is also simple to update the structure for removals, and moreover the code does not depend on whether the reference was from a `child` pointer or a `brother` pointer. Still the `hook` information must be properly updated, for example if `p->b` becomes the `child` of node `x` we would use the following code:

```
x->child = p->b;
p->b->hook = &(x->child);
```

It is recommended to allocate an array containing all the structures that will be necessary, this avoids having to call the function `malloc` for each structure. This is possible because we know from the start that at most `2*((2*n)+1)` nodes are necessary, which includes leafs, the root and the space for the sentinel. Also this allows us to verify the correct implementation of the structure by referring to positions in this structure. The `root` node should be store in position 0. The sentinel node in position 1. The remaining nodes are stored in incremental positions at the time that they are first necessary. Whenever inserting a new suffix creates an internal node and a leaf the internal node is inserted first and the leaf second. The following functions can be used to print the information about a node, by converting the pointers back into array positions.

```
int ptr2loc(node v, node A)
{
  int r;
  r = -1;
  if(NULL != v)
    r = ((size_t) v - (size_t) A) / sizeof(struct node);
```

```
  return (int)r;
}

void shownode(node v)
{
  if(NULL == v)
    printf("NULL\n");
  else {
    printf("node: %d ", ptr2loc(v, root));
    printf("head: %d ", v->head);
    printf("sdep: %d ", v->sdep);
    printf("child: %d ", ptr2loc(v->child, root));
    printf("brother: %d ", ptr2loc(v->brother, root));
    printf("slink: %d", ptr2loc(v->slink, root));
    printf("\n");
  }
}
```

A simple outline of Ukkonen's algorithm is the following, slightly different from section 6.1:

```
  i = 0;
  while(i <= 2*n) {
    printf("Letter %c\n", '\0' == T[i] ? '$' : T[i]);
    while(!DescendQ(p, T[i])) {
        a += AddLeaf(p, &(root[a]), i);
        SuffixLink(p);
      }
    Descend(p, T[i]);
    i++;
  }
```

This code is used to build the suffix tree, where `T` stores the text and terminates with a `'\0'` character, the size of this text is stored in the variable `n`. For every character `T[i]` that needs to be inserted we create nodes and follow suffix links until we find a position where it is possible to descend by the letter `T[i]`. The variable `p` represents a point, i.e., a position between two letters of a given branch. The following structure is a possible implementation of this concept:

```
typedef struct point* point;
```

```
struct point {
  node a; /**< node above */
  node b; /**< node bellow */
  int s; /**< String-Depth */
};
```

The `DescendQ` function returns true when it is possible to descend from `p` with `T[i]`. Then `Descend` function actually descends with the given letter. The `AddLeaf` function creates a new leaf and inserts it into the tree, in this process it might also be necessary to create a new internal node. The `SuffixLink` alters the point `p`, so that it points to the suffix link of the current point. This function must implement the skip/count trick of section 6.1.3.

During the execution of the `AddLeaf` function internal nodes and leafs are, possibly, created. If the `AddLeaf` function needs to create a new internal node its `head` value is copied from the node bellow the edge being split. In fact the new node replaces the bellow node in whichever linked list, and position, it happens to occur. The existing node, or leaf, therefore becomes the `child` of the new node. When a leaf is created its `child` and `slink` fields are set to `NULL` and this setting will not change during the rest of the algorithm. The new leaf becomes the first element of its linked list.

An important detail to be mindful about is that the `AddLeaf` function should not alter `p`, the reason for this is that the `SuffixLink` uses the value in the `slink` field, and for recently created nodes this value is not yet available. In fact assigning this value, for a new node, is another crucial detail, it should be set by `SuffixLink` if `p` is an internal node, otherwise it should be set to the next new internal node, created on the next call of `AddLeaf`, this issue is referred in Lemma 6.1.1.

To verify the correct construction of the suffix tree the program should output information about the nodes it creates in `AddLeaf`. Whenever an internal node is create the program should output the string `'Internal '` followed by a call to the function `shownode`. Whenever a leaf is created the program should output the string `'Leaf '` followed by a call to the `shownode` function.

After building the suffix tree it is necessary to traverse it. For this process start at the root and at each node select the child that starts by lexicographic smallest letter, ignoring terminators. Stop the process once a string depth of `n` is obtained. At this point the algorithm should print the string found. The following command may be useful.

```
printf("%.*s\n", n, &T[p->b->head]);
```

As a final verification procedure after the tree is completely built the program should print all the nodes, in their array order. This time the information about the suffix-link pointers should be correct. The following code snippet gives an implementation of this scan, assuming the variable `a` indicates the actual number of nodes in the tree.

```
printf("Final version\n");
i = 0;
while(i < a){
  shownode(&(root[i]));
  i++;
}
```

1.2 Specification

To automatically validate the index we use the following conventions. The binary is executed with the following command:

`./project < in > out`

The file `in` contains the input commands that we will describe next. The output is stored in a file named `out`. The input and output must respect the specification below precisely. Note that your program should **not** open or close files, instead it should read information from `stdin` and write to `stdout`. The output file will be validated against an expected result, stored in a file named `check`, with the following command:

`diff out check`

This command should produce no output, thus indicating that both files are identical.

The format of the input file is the following. The first line contains a single integer, n, i.e., the size of the string that follows. Then there is a single space character and then the characters of string `T`.

The input contains no more data.

The output should consist of `printf` statements indicated in the code given above. Here is a brief review. The string `Letter` and the corresponding letter being processed. The string `Internal` when an internal node is created. The string `Leaf` when a leaf is created. The string `Final version` followed by a sequence of `shownode` calls.

1.3 Sample Behaviour

The following examples show the expected `output` for the given `input`. These files are available on the course web page.

input 1

```
3 TAA
```

output 1

```
Letter T
Leaf node: 2 head: 0 sdep: 7 child: -1 brother: -1 slink: -1
Letter A
Leaf node: 3 head: 1 sdep: 6 child: -1 brother: 2 slink: -1
Letter A
Letter T
Internal node: 4 head: 1 sdep: 1 child: 3 brother: 2 slink: -1
Leaf node: 5 head: 2 sdep: 5 child: -1 brother: 3 slink: -1
Letter A
Letter A
Letter $
Internal node: 6 head: 0 sdep: 3 child: 2 brother: -1 slink: -1
Leaf node: 7 head: 3 sdep: 4 child: -1 brother: 2 slink: -1
Internal node: 8 head: 1 sdep: 2 child: 3 brother: -1 slink: -1
Leaf node: 9 head: 4 sdep: 3 child: -1 brother: 3 slink: -1
Leaf node: 10 head: 5 sdep: 2 child: -1 brother: 5 slink: -1
Leaf node: 11 head: 6 sdep: 1 child: -1 brother: 4 slink: -1
AAT
Final version
node: 0 head: 0 sdep: 0 child: 11 brother: -1 slink: 1
node: 1 head: 0 sdep: -1 child: 0 brother: -1 slink: -1
node: 2 head: 0 sdep: 7 child: -1 brother: -1 slink: -1
node: 3 head: 1 sdep: 6 child: -1 brother: -1 slink: -1
node: 4 head: 1 sdep: 1 child: 10 brother: 6 slink: 0
node: 5 head: 2 sdep: 5 child: -1 brother: 8 slink: -1
node: 6 head: 0 sdep: 3 child: 7 brother: -1 slink: 8
node: 7 head: 3 sdep: 4 child: -1 brother: 2 slink: -1
node: 8 head: 1 sdep: 2 child: 9 brother: -1 slink: 4
node: 9 head: 4 sdep: 3 child: -1 brother: 3 slink: -1
node: 10 head: 5 sdep: 2 child: -1 brother: 5 slink: -1
node: 11 head: 6 sdep: 1 child: -1 brother: 4 slink: -1
```

input 2

```
5 TAAAA
```

output 2

```
Letter T
Leaf node: 2 head: 0 sdep: 11 child: -1 brother: -1 slink: -1
Letter A
Leaf node: 3 head: 1 sdep: 10 child: -1 brother: 2 slink: -1
Letter A
Letter A
Letter A
Letter T
Internal node: 4 head: 1 sdep: 3 child: 3 brother: 2 slink: -1
Leaf node: 5 head: 2 sdep: 9 child: -1 brother: 3 slink: -1
Internal node: 6 head: 1 sdep: 2 child: 4 brother: 2 slink: -1
Leaf node: 7 head: 3 sdep: 8 child: -1 brother: 4 slink: -1
Internal node: 8 head: 1 sdep: 1 child: 6 brother: 2 slink: -1
Leaf node: 9 head: 4 sdep: 7 child: -1 brother: 6 slink: -1
Letter A
Letter A
Letter A
Letter A
Letter $
Internal node: 10 head: 0 sdep: 5 child: 2 brother: -1 slink: -1
Leaf node: 11 head: 5 sdep: 6 child: -1 brother: 2 slink: -1
Internal node: 12 head: 1 sdep: 4 child: 3 brother: -1 slink: -1
Leaf node: 13 head: 6 sdep: 5 child: -1 brother: 3 slink: -1
Leaf node: 14 head: 7 sdep: 4 child: -1 brother: 5 slink: -1
Leaf node: 15 head: 8 sdep: 3 child: -1 brother: 7 slink: -1
Leaf node: 16 head: 9 sdep: 2 child: -1 brother: 9 slink: -1
Leaf node: 17 head: 10 sdep: 1 child: -1 brother: 8 slink: -1
AAAAT
Final version
node: 0 head: 0 sdep: 0 child: 17 brother: -1 slink: 1
node: 1 head: 0 sdep: -1 child: 0 brother: -1 slink: -1
node: 2 head: 0 sdep: 11 child: -1 brother: -1 slink: -1
node: 3 head: 1 sdep: 10 child: -1 brother: -1 slink: -1
node: 4 head: 1 sdep: 3 child: 14 brother: -1 slink: 6
node: 5 head: 2 sdep: 9 child: -1 brother: 12 slink: -1
node: 6 head: 1 sdep: 2 child: 15 brother: -1 slink: 8
node: 7 head: 3 sdep: 8 child: -1 brother: 4 slink: -1
node: 8 head: 1 sdep: 1 child: 16 brother: 10 slink: 0
node: 9 head: 4 sdep: 7 child: -1 brother: 6 slink: -1
```

```
node: 10 head: 0 sdep: 5 child: 11 brother: -1 slink: 12
node: 11 head: 5 sdep: 6 child: -1 brother: 2 slink: -1
node: 12 head: 1 sdep: 4 child: 13 brother: -1 slink: 4
node: 13 head: 6 sdep: 5 child: -1 brother: 3 slink: -1
node: 14 head: 7 sdep: 4 child: -1 brother: 5 slink: -1
node: 15 head: 8 sdep: 3 child: -1 brother: 7 slink: -1
node: 16 head: 9 sdep: 2 child: -1 brother: 9 slink: -1
node: 17 head: 10 sdep: 1 child: -1 brother: 8 slink: -1
```

input 3

```
5 TACTG
```

output 3

```
Letter T
Leaf node: 2 head: 0 sdep: 11 child: -1 brother: -1 slink: -1
Letter A
Leaf node: 3 head: 1 sdep: 10 child: -1 brother: 2 slink: -1
Letter C
Leaf node: 4 head: 2 sdep: 9 child: -1 brother: 3 slink: -1
Letter T
Letter G
Internal node: 5 head: 0 sdep: 1 child: 2 brother: -1 slink: -1
Leaf node: 6 head: 3 sdep: 8 child: -1 brother: 2 slink: -1
Leaf node: 7 head: 4 sdep: 7 child: -1 brother: 4 slink: -1
Letter T
Letter A
Letter C
Letter T
Letter G
Letter $
Internal node: 8 head: 0 sdep: 5 child: 2 brother: -1 slink: -1
Leaf node: 9 head: 5 sdep: 6 child: -1 brother: 2 slink: -1
Internal node: 10 head: 1 sdep: 4 child: 3 brother: 5 slink: -1
Leaf node: 11 head: 6 sdep: 5 child: -1 brother: 3 slink: -1
Internal node: 12 head: 2 sdep: 3 child: 4 brother: 10 slink: -1
Leaf node: 13 head: 7 sdep: 4 child: -1 brother: 4 slink: -1
Internal node: 14 head: 3 sdep: 2 child: 6 brother: 8 slink: -1
Leaf node: 15 head: 8 sdep: 3 child: -1 brother: 6 slink: -1
Internal node: 16 head: 4 sdep: 1 child: 7 brother: 12 slink: -1
```

```
Leaf node: 17 head: 9 sdep: 2 child: -1 brother: 7 slink: -1
Leaf node: 18 head: 10 sdep: 1 child: -1 brother: 16 slink: -1
ACTGT
Final version
node: 0 head: 0 sdep: 0 child: 18 brother: -1 slink: 1
node: 1 head: 0 sdep: -1 child: 0 brother: -1 slink: -1
node: 2 head: 0 sdep: 11 child: -1 brother: -1 slink: -1
node: 3 head: 1 sdep: 10 child: -1 brother: -1 slink: -1
node: 4 head: 2 sdep: 9 child: -1 brother: -1 slink: -1
node: 5 head: 0 sdep: 1 child: 14 brother: -1 slink: 0
node: 6 head: 3 sdep: 8 child: -1 brother: -1 slink: -1
node: 7 head: 4 sdep: 7 child: -1 brother: -1 slink: -1
node: 8 head: 0 sdep: 5 child: 9 brother: -1 slink: 10
node: 9 head: 5 sdep: 6 child: -1 brother: 2 slink: -1
node: 10 head: 1 sdep: 4 child: 11 brother: 5 slink: 12
node: 11 head: 6 sdep: 5 child: -1 brother: 3 slink: -1
node: 12 head: 2 sdep: 3 child: 13 brother: 10 slink: 14
node: 13 head: 7 sdep: 4 child: -1 brother: 4 slink: -1
node: 14 head: 3 sdep: 2 child: 15 brother: 8 slink: 16
node: 15 head: 8 sdep: 3 child: -1 brother: 6 slink: -1
node: 16 head: 4 sdep: 1 child: 17 brother: 12 slink: 0
node: 17 head: 9 sdep: 2 child: -1 brother: 7 slink: -1
node: 18 head: 10 sdep: 1 child: -1 brother: 16 slink: -1
```

input 4

```
8 TATATATA
```

output 4

```
Letter T
Leaf node: 2 head: 0 sdep: 17 child: -1 brother: -1 slink: -1
Letter A
Leaf node: 3 head: 1 sdep: 16 child: -1 brother: 2 slink: -1
Letter T
Letter A
Letter T
Letter A
Letter T
Letter A
Letter T
```

```
Letter A
Letter T
Letter A
Letter T
Letter A
Letter T
Letter A
Letter $
Internal node: 4 head: 0 sdep: 14 child: 2 brother: -1 slink: -1
Leaf node: 5 head: 2 sdep: 15 child: -1 brother: 2 slink: -1
Internal node: 6 head: 1 sdep: 13 child: 3 brother: 4 slink: -1
Leaf node: 7 head: 3 sdep: 14 child: -1 brother: 3 slink: -1
Internal node: 8 head: 0 sdep: 12 child: 4 brother: -1 slink: -1
Leaf node: 9 head: 4 sdep: 13 child: -1 brother: 4 slink: -1
Internal node: 10 head: 1 sdep: 11 child: 6 brother: 8 slink: -1
Leaf node: 11 head: 5 sdep: 12 child: -1 brother: 6 slink: -1
Internal node: 12 head: 0 sdep: 10 child: 8 brother: -1 slink: -1
Leaf node: 13 head: 6 sdep: 11 child: -1 brother: 8 slink: -1
Internal node: 14 head: 1 sdep: 9 child: 10 brother: 12 slink: -1
Leaf node: 15 head: 7 sdep: 10 child: -1 brother: 10 slink: -1
Internal node: 16 head: 0 sdep: 8 child: 12 brother: -1 slink: -1
Leaf node: 17 head: 8 sdep: 9 child: -1 brother: 12 slink: -1
Internal node: 18 head: 1 sdep: 7 child: 14 brother: 16 slink: -1
Leaf node: 19 head: 9 sdep: 8 child: -1 brother: 14 slink: -1
Internal node: 20 head: 0 sdep: 6 child: 16 brother: -1 slink: -1
Leaf node: 21 head: 10 sdep: 7 child: -1 brother: 16 slink: -1
Internal node: 22 head: 1 sdep: 5 child: 18 brother: 20 slink: -1
Leaf node: 23 head: 11 sdep: 6 child: -1 brother: 18 slink: -1
Internal node: 24 head: 0 sdep: 4 child: 20 brother: -1 slink: -1
Leaf node: 25 head: 12 sdep: 5 child: -1 brother: 20 slink: -1
Internal node: 26 head: 1 sdep: 3 child: 22 brother: 24 slink: -1
Leaf node: 27 head: 13 sdep: 4 child: -1 brother: 22 slink: -1
Internal node: 28 head: 0 sdep: 2 child: 24 brother: -1 slink: -1
Leaf node: 29 head: 14 sdep: 3 child: -1 brother: 24 slink: -1
Internal node: 30 head: 1 sdep: 1 child: 26 brother: 28 slink: -1
Leaf node: 31 head: 15 sdep: 2 child: -1 brother: 26 slink: -1
Leaf node: 32 head: 16 sdep: 1 child: -1 brother: 30 slink: -1
ATATATAT
Final version
node: 0 head: 0 sdep: 0 child: 32 brother: -1 slink: 1
node: 1 head: 0 sdep: -1 child: 0 brother: -1 slink: -1
```

```
node: 2 head: 0 sdep: 17 child: -1 brother: -1 slink: -1
node: 3 head: 1 sdep: 16 child: -1 brother: -1 slink: -1
node: 4 head: 0 sdep: 14 child: 5 brother: -1 slink: 6
node: 5 head: 2 sdep: 15 child: -1 brother: 2 slink: -1
node: 6 head: 1 sdep: 13 child: 7 brother: -1 slink: 8
node: 7 head: 3 sdep: 14 child: -1 brother: 3 slink: -1
node: 8 head: 0 sdep: 12 child: 9 brother: -1 slink: 10
node: 9 head: 4 sdep: 13 child: -1 brother: 4 slink: -1
node: 10 head: 1 sdep: 11 child: 11 brother: -1 slink: 12
node: 11 head: 5 sdep: 12 child: -1 brother: 6 slink: -1
node: 12 head: 0 sdep: 10 child: 13 brother: -1 slink: 14
node: 13 head: 6 sdep: 11 child: -1 brother: 8 slink: -1
node: 14 head: 1 sdep: 9 child: 15 brother: -1 slink: 16
node: 15 head: 7 sdep: 10 child: -1 brother: 10 slink: -1
node: 16 head: 0 sdep: 8 child: 17 brother: -1 slink: 18
node: 17 head: 8 sdep: 9 child: -1 brother: 12 slink: -1
node: 18 head: 1 sdep: 7 child: 19 brother: -1 slink: 20
node: 19 head: 9 sdep: 8 child: -1 brother: 14 slink: -1
node: 20 head: 0 sdep: 6 child: 21 brother: -1 slink: 22
node: 21 head: 10 sdep: 7 child: -1 brother: 16 slink: -1
node: 22 head: 1 sdep: 5 child: 23 brother: -1 slink: 24
node: 23 head: 11 sdep: 6 child: -1 brother: 18 slink: -1
node: 24 head: 0 sdep: 4 child: 25 brother: -1 slink: 26
node: 25 head: 12 sdep: 5 child: -1 brother: 20 slink: -1
node: 26 head: 1 sdep: 3 child: 27 brother: -1 slink: 28
node: 27 head: 13 sdep: 4 child: -1 brother: 22 slink: -1
node: 28 head: 0 sdep: 2 child: 29 brother: -1 slink: 30
node: 29 head: 14 sdep: 3 child: -1 brother: 24 slink: -1
node: 30 head: 1 sdep: 1 child: 31 brother: 28 slink: 0
node: 31 head: 15 sdep: 2 child: -1 brother: 26 slink: -1
node: 32 head: 16 sdep: 1 child: -1 brother: 30 slink: -1
```

2 Grading

The mooshak system is configured to a total 40 points. The project accounts for 4.0 values of the final grade. Hence to obtain the contribution of the project to the final grade divide the number of points by 10. To obtain a grading in an absolute scale to 20 divide the number of points by 2.

Each test has a specific set of points. The first four tests correspond to the input output examples given in this script. These tests are public

and will be returned back by the system. The tests numbered from 5 to 12 correspond to increasingly harder test cases, brief descriptions are given by the system. Tests 13 and 14 are verified by the valgrind[1] tool. Test 13 checks for the condition `ERROR SUMMARY: 0 errors from 0 contexts` and test 14 for the condition `All heap blocks were freed -- no leaks are possible`. Test 15 to 17 are verified by the lizzard[2] tool, the test passes if the `No thresholds exceeded` message is given. Test 15 uses the arguments `-T cyclomatic_complexity=15`; test 16 the argument `-T length=150`; test 17 the argument `-T parameter_count=9 -T token_count=500`. To obtain the score of tests from 13 to 17 must it is necessary obtain the correct output, besides the conditions just described.

The mooshak system accepts the C programming language, click on `Help` button for the respective compiler. Projects that do not compile in the mooshak system will be graded 0. Only the code that compiles in the mooshak system will be considered, commented code will not be considered for evaluation.

Submissions to the mooshak system should consist of a single file. The system identifies the language through the file extension, an extension `.c` means the C language. The compilation process should produce absolutely no errors or warnings, otherwise the file will not compile. The resulting binary should behave exactly as explained in the specification section. Be mindful that `diff` will produce output even if a single character is different, such as a space or a newline.

Notice that you can submit to mooshak several times, but there is a 10 minute waiting period before submissions. You are strongly advised to submit several times and as early as possible. Only the last version is considered for grading purposes, all other submissions are ignored. There will be **no** deadline extensions. Submissions by email will **not** be accepted.

References

[1] Gusfield, D. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge Univ Press, 1997.

[1] `https://www.valgrind.org/`

[2] `https://github.com/terryyin/lizard`