# Computer Science I — CSCI-141
# DNA Toolkit — Lab 9

11/06/2023

## 1 Problem

A single strand of DNA contains a sequence of nucleobases: cytosine (C), guanine (G), adenine (A), and thymine (T). DNA mutations take many forms, including:

- substitution: a single base is replaced with another
- insertion: a sequence of bases is inserted consecutively within an existing strand
- deletion: a portion of an existing strand is removed
- duplication: a portion of an existing strand is repeated immediately following itself

After taking a course in genetics and learning about different DNA mutations, you decide that it might be advantageous when modeling these mutations to represent a single DNA strand as a linked sequence. DNA mutations may involve substantial insertions and deletions at various locations throughout the DNA strand, and a *linked sequence data structure* can handle such modifications well. Moreover, so that you can track the mutations over time, you choose to use *non-destructive linked sequence* operations that return a new sequence representing the result after mutation, and retain the input sequences in their original form.

### 1.1 Problem-solving Session (15%)

Work in a team of three or four students as determined by your instructor. Each team will work together to complete the problem-solving activities.

Recall the definitions of the linked data structure from lecture.

```
from dataclasses import dataclass
from typing import Any, Union

@dataclass( frozen=True )
class FrozenNode:
    """
    An immutable link node containing a value and a link to the next node
    """
    value: Any
    next: Union[ "FrozenNode" , None ]
```

For the problem-solving session your functions **must be recursive**.

Note also that *immutable node sequences* are **naked**; the sequences do not have an *encapsulating dataclass*.

Use the *structural recursive template* to guide you. Consider what base cases will be needed. For the recursive case(s), consider how you can combine the first element with the result of a recursive call that solves the problem for the remaining elements.

You may look up, incorporate and reuse linked sequence functions covered in lecture, or directly manipulate the linked sequence structure.

First be sure to write correct import statements needed for the problems.

1. Write a function, `convert_to_nodes()`, that takes one parameter as input, a single DNA strand represented as a string (e.g. `"ATGGTCCAAT"`). This function builds and returns a node-based linked sequence representation of the DNA strand.

2. Write a function, `is_match()`, that takes two parameters, a first DNA linked sequence, and a second DNA linked sequence. This function computes whether the two sequences are exact matches. To be exact matches, they must be the same length and the base values must match at every position.

3. Write a function, `insert_seq()`, that takes three parameters: a first DNA linked sequence, a second DNA linked sequence, and an index. This function implements insertion functionality; it constructs and returns a new linked sequence that contains the second DNA sequence inserted within the first sequence beginning at the specified index.

   For example, if `dna_seq1` represented as a string is `"ATCCCG"` and `dna_seq2` represented as a string is `"GTAA"`, and `index == 3`, then the function returns a new linked sequence whose string representation would be the string `"ATCGTAACCG"`.

   This function must be *non-destructive*. That is, the input sequences must not be modified. It must create a new sequence. You may assume the input index is non-negative. However, the code must detect if the index is in range. If the index is outside the range of legal insertion, the code should `raise IndexError("invalid insertion index")`.

   A `concatenate` function will be helpful for this problem. *Assume such a function exists*, and the call `concatenate( seq1, seq2 )` will return a new *linked sequence* that is the concatenated result.

## 2 In-lab Session (10%)

Work on these tasks during the in-lab period. At the end, have an SLI or the instructor check your progress. Students who have put in a reasonable amount of effort will receive full credit for the in-lab. Students who do not demonstrate reasonable effort, leave early, or do not get verification, will not get credit.

Tasks:

1. *Read the entire lab carefully.*

2. Create a new project and open a new file, `dna.py`, for development.

3. Download the provided code zip file, which contains a module with extra linked list functions and a tester file, `dna_tester.py`; here is the link:

   `http://www.cs.rit.edu/~csci141/Labs/09/Provided.zip`.

   The zip file contains the `node_types.py` file from the lecture, plus `immutable_extra.py`, which contains these additional functions to use in your solution:

   - `concatenate(head1, head2)` produces the sequence resulting from the concatenation of the two linked lists: `head1` and `head2`.
   - `remove_at(index, head)` produces the sequence resulting from the removal of the value at the given index from `head`.

4. Write the function to convert a string into a linked node sequence. Use the provided test module to confirm that it functions correctly.

5. Use the rest of the in-lab to implement and test functions from problem-solving.

Zip your `dna.py` file into `lab09.zip` and upload the zip file to the In-lab assignment dropbox by the deadline. Do not zip any other files.

## 3 Implementation (75%)

Each student will *individually implement* and submit their own solution to the problem as a Python program named `dna.py`.

### Requirements

The provided module, `dna_tester.py`, will test your module.

Your module should not have a `main` function because your program will not be used as a main program. Your module provides functionality that enlarges and extends the `immutable_extra` functionality.

You may use recursion or iteration in your solution.

You also may use any functionality provided by the `immutable_extra` module. You might find that functions you write can utilize other functions you have written!

Your program must implement all the functions listed below.

**Note: all functions must operate on linked sequences. You may not do your work on standard Python sequences, such as lists, strings or tuples. You may not convert the data to a Python sequence, analyze or manipulate the Python sequence, and convert back to a linked node sequence. You will receive no credit if you violate this restriction.**

*Do NOT copy any code from the* `immutable_extra.py` *file into the* `dna.py` *file. You must use the* **import** *mechanism to use the functions from the module.*

1. `convert_to_nodes( dna_string )`

   Parameters:
   - `dna_string`: a string of characters corresponding to DNA bases.

   Return value: a linked-node data structure representing the input DNA sequence. *Each character of the input string is represented as one node in the sequence.*

   Behavior:
   - If the input string is empty, the linked-node data structure is the value `None`.
   - The $k^{th}$ character in the input string must be represented by the $k^{th}$ node in the sequence. Indexing is *zero-based.*

2. `convert_to_string( dna_seq )`

   Parameters:
   - `dna_seq`: a linked sequence in which each node contains as data one character representing one base of a DNA sequence.

   Return value: a string.

   Behavior:
   - If the input sequence is empty, return the empty string.
   - The data from the $k^{th}$ node in the input sequence must correspond to the $k^{th}$ character of the returned string.

3. `length_rec( dna_seq )`

   Parameters:
   - `dna_seq`: a linked sequence in which each node contains as data one character representing one base of a DNA sequence.

   Return value: an integer.

   Behavior:
   - If the input sequence is empty, return 0.
   - Compute the length of `dna_seq` recursively.

4. `is_match( dna_seq1, dna_seq2 )`

   Parameters:
   - `dna_seq1`: a first linked sequence with nodes representing a DNA sequence.
   - `dna_seq2`: a second linked sequence with nodes representing a DNA sequence.

   Return value: boolean `True` or `False`.

   Behavior:
   - This function conducts an element by element comparison of the two sequences. To be a match and return `True`, the two sequences must be of equal length, and the elements at each index must be the same DNA base. If the sequences are different length, or if there is any index at which the corresponding elements are not identical, the function returns `False`.

5. `is_pairing( dna_seq1, dna_seq2 )`

   Parameters:
   - `dna_seq1`: a first linked sequence with nodes representing a DNA sequence.
   - `dna_seq2`: a second linked sequence with nodes representing a DNA sequence.

   Return value: boolean `True` or `False`.

   Behavior:
   - This function performs an element by element comparison of the two sequences. To be a valid pairing and return `True`, the two sequences must be of equal length, and the elements at each index must be a valid *DNA base pairing*. That is, for each index, the elements of the two sequences at that index must be a valid DNA base pair.
     If the sequences are different length, or if there is any index at which the corresponding elements are not a valid pair, the function returns `False`.
     **Valid base pairings** are (A with T) and (G with C). Note that the valid pairings are *symmetric*. That is, an A in the first sequence can be paired with a T in the second sequence. Equally valid is a T in the first sequence paired with an A in the second sequence.

6. `substitute( dna_seq1, idx, base )`

   Parameters:
   - `dna_seq1`: the source sequence for the substitution mutation.
   - `idx`: the index at which the substitution occurs.
   - `base`: the new base to be substituted at the specified index.

   Return value: a *new* linked sequence that represents the DNA strand after the substitution mutation has occurred.

   Behavior:
   - You may assume `idx >= 0`. However, `idx` may be out of range. If `idx` is out of range, the function should `raise` an `IndexError`. See the provided module

immutable_extra.py for examples of raising an IndexError.

- **This function must be *non-destructive* and must not modify the input sequence.**

7. `insert_seq( dna_seq1, dna_seq2, idx )`

   Parameters:

   - `dna_seq1`: the first source sequence for insertion mutation.
   - `dna_seq2`: the second sequence to be inserted into the first.
   - `idx`: the index before which insertion must occur (similar to `insert_before_index`).

   Return value: a *new* linked sequence that represents the resulting DNA strand after `dna_seq2` has been inserted in its entirety just before index `idx` within `dna_seq1`.

   Behavior:

   - You may assume `idx >= 0`. However, `idx` may be out of range. If `idx` is out of range, the function should `raise` an `IndexError`. Note that `idx` equal to `0` is always a valid insertion index, even if `dna_seq1` is empty. An `idx` value equal to the length of `dna_seq1` is also valid, and corresponds to extending `dna_seq1` with `dna_seq2`.
   - The function inserts the entire `dna_seq2` just before the index specified. For example, suppose `dna_seq1` represents the DNA sequence "ATCG", `dna_seq2` represents the DNA sequence "AGCCA" and `idx = 2`. The returned result of a call to `insert_seq( dna_seq1, dna_seq2, 2 )` would be a new structure that represents the sequence "ATAGCCACG".
   - **This function must be *non-destructive*; it must not modify either of the input sequences.**

8. `delete_seq( dna_seq, idx, segment_size )`

   Parameters:

   - `dna_seq`: the source sequence for the deletion mutation.
   - `idx`: the index at which deletion begins.
   - `segment_size`: the number of elements to be deleted.

   Return value: a *new* linked sequence that represents the resulting DNA strand after the specified segment of elements has been removed from `dna_seq`.

   Behavior:

   - You may assume `idx >= 0`. You may also assume that `segment_size >= 0`.
   - If `segment_size` is equal to `0`, deletion succeeds by default, *even if* `idx` *is out of range*, and should return the original sequence.
   - Otherwise, if the combination of `idx` and `segment_size` is out of range, the function should `raise` an `IndexError`.
   - The deleted segment begins at the specified index. For example, suppose `dna_seq` represents the DNA sequence "ATACAGAGT". The returned result of a call to `delete_seq( dna_seq, 4, 3 )` would be a new structure that represents the sequence "ATACGT".

- **This function must be *non-destructive* and must not modify the input sequence.**

9. `duplicate_seq( dna_seq, idx, segment_size )`

   Parameters:
   - `dna_seq`: the source sequence for the duplication mutation.
   - `idx`: the start index of the segment to duplicate.
   - `segment_size`: the number of elements to duplicate.

   Return value: a *new* linked sequence that represents the resulting DNA strand after the specified segment from `dna_seq` has been duplicated.

   Behavior:
   - You may assume `idx >= 0`. You may also assume that `segment_size >= 0`.
   - If `segment_size` is equal to `0`, duplication succeeds by default, *even if* `idx` *is out of range*, and should return the original sequence.
   - Otherwise, if the combination of `idx` and `segment_size` is out of range, the function should `raise` an `IndexError`.
   - The duplicated segment should immediately follow the original segment itself. For example, suppose `dna_seq` represents the DNA sequence "ATACAGAGT". The returned result of a call to `duplicate_seq( dna_seq, 4, 3 )` would be a new structure that represents the sequence "ATACAGAAGAGT".
   - **This function must be *non-destructive* and must not modify the input sequence.**

**Note:** The requirement that a function is *non-destructive* does not necessarily imply that the code will have to create an entirely new sequence. Consider the `insert_before_index` function in `immutable_extra.py` as an example. The `insert_before_index` function creates new nodes until it reaches the point where it inserts the new node. After that point, the function retains the original sequence as is.

## 4 Testing

Use the file, `dna_tester.py`, to confirm that your functions work correctly. This file has two primary functions:

- `test_individual`: this function tests individual functions with numbered tests, `test1`, `test2`, . . . . Each function can be tested independently by commenting out calls to test functions that you haven't implemented yet. Uncomment calls as you implement them.

- `test_all`: this function tests all the required functions and serves as a complete test suite or regression test. Leave this call commented out until you have completed all functions.

## 5 Grading

The 75% implementation grade is based on this rubric:

- 5%: `convert_to_nodes`
- 5%: `convert_to_string`
- 5%: `length_rec`
- 8%: `is_pairing`
- 8%: `is_match`
- 11%: `insert_seq`
- 11%: `substitute`
- 11%: `delete_seq`
- 11%: `duplicate_seq`
- *10% off the top*: Failure to follow course style and documentation guidelines, or to follow submission directions
- ***30% off the top*: Code was copied from `immutable_extra.py` into `dna.py` instead of imported.**

## 6 Submission

Zip your `dna.py` file along with the provided python files, name it `lab09.zip`, and submit the zip file to the appropriate lab dropbox by the due date. ZIP is the only acceptable archive format (i.e. do not use 7-Zip, WinRAR, etc.).