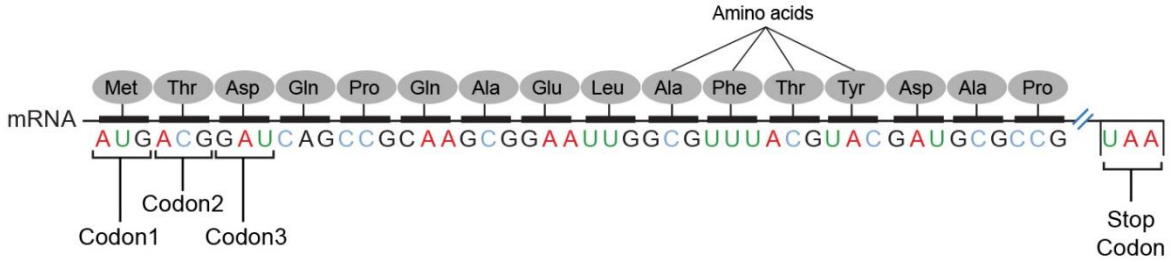
ENGR 105 – Introduction to Scientific Computing

Assignment 5

Due by 11:59 pm on Weds. 2/16/2022 on Gradescope

Problem 1 (40 pts): Deoxyribonucleic acid (DNA) contains polynucleotide chains that constitute the genetic code for organisms and many viruses. Messenger ribonucleic acid (mRNA) transcribes the DNA in the process of protein synthesis. mRNA can be represented by the series of four nucleotide bases uracil (U), cytosine (C), adenine (A), and guanine (G). Triplets of these bases are known as codons, which transcribe to specific amino acids and, in some cases, instructions for transcription to stop.



Sections of the amino acid sequence from mRNA are important to those studying viruses, pain, and hereditary diseases (just to name a few applications).

The mapping from the bases to amino acid abbreviations for mRNA is shown in the table below. A color-coded table is available here.

First base	Second base								Third base
	U (1)		C (2)		A (3)		G (4)		
U (1)	UUU	Phe	UCU	Ser	UAU	Tyr	UGU	Cys	U (1)
	UUC		UCC		UAC		UGC		C (2)
	UUA	Leu	UCA		UAA	STOP	UGA	STOP	A (3)
	UUG		UCG		UAG		UGG	Trp	G (4)
C (2)	CUU	Leu	CCU	Pro	CAU	His	CGU	Arg	U (1)
	CUC		CCC		CAC		CGC		C (2)
	CUA		CCA		CAA	Gln	CGA		A (3)
	CUG		CCG		CAG		CGG		G (4)
A (3)	AUU	Ile	ACU	Thr	AAU	Asn	AGU	Ser	U (1)
	AUC		ACC		AAC		AGC		C (2)
	AUA		ACA		AAA	Lys	AGA	Arg	A (3)
	AUG	Met	ACG		AAG		AGG		G (4)
G (4)	GUU	Val	GCU	Ala	GAU	Asp	GGU	Gly	U (1)
	GUC		GCC		GAC		GGC		C (2)
	GUA		GCA		GAA	Glu	GGA		A (3)
	GUG		GCG		GAG		GGG		G (4)

For example, both UUU [1, 1, 1] and UUC [1, 1, 2] would produce phenylalanine (abbreviated “Phe”). As another example, UGA [1, 4, 3] would encode “STOP”.

You are provided with an mRNA sequence of the coronavirus 2 isolate Wuhan-Hu-1 derived from the complete genome as file `RNA_sequence.mat`. This sequence is a 29,901-index long row vector that contains numbers that represent mRNA bases using the mapping U = 1, C = 2, A = 3, and G = 4.

Devise a function that produces the amino acid sequence for an input RNA sequence. The function should have the following function declaration.

```
function amino_sequence = base2amino(RNA_sequence)
```

Input `RNA_sequence` is a row vector and you may assume that `mod(length(RNA_sequence),3) = 0`. Codon transcription should start at `RNA_sequence(1)`. Output `amino_sequence` is a character string that contains the amino acid abbreviations separated by hyphens. The first few and last few amino acid sequences for `RNA_sequence.mat` are as follows.

First few amino amino acids: `STOP-Phe-Pro-Asn-Met-Glu-Gly-Ser-Ile-Val-Trp…`
Last few amino acids: `…Cys-Phe-Phe-Phe-Phe-Phe-Phe-Phe-Phe-Phe-Phe`

A reference `amino_sequence` corresponding to `RNA_sequence.mat` is provided as file `amino_sequence_reference.mat`.

Devise a script that loads `RNA_sequence.mat`, runs `base2amino`, and compares the output of your `base2amino` function to that of `amino_sequence_reference.mat`.

A few helpful coding hints follow.

The RNA sequence can be imported into a MATLAB script using the following code, which results in a variable named `RNA_sequence`. The `amino_sequence_reference.mat` data can be imported similarly.

```
load 'RNA_sequence.mat'
```

Character arrays can be concatenated similar to vectors. Consider a variable named `result`, which resolves to the character array `'-Some text-Some more text'` for the code below.

```
result = []
result = [result, '-Some text']
result = [result, '-Some more text']
```

Two character arrays can be compared using code similar to the following.

```
compare = sum('abcdef' ~= 'abgdeh')
```

The variable `compare` is zero if the character arrays are identical and nonzero if they differ.

Upload your `base2amino` function and the script you used to launch the function and compare results. Identify the function and script names in your README.txt.

Problem 2 (60 pts): Have you ever received a call from a telemarketing company, obtained directions using google maps, or played an online card game? These are just some of many situations in which sorting is productively (or in the case of telemarketing, unproductively) used in the real world.

There are many sorting algorithms, each with varying speed, memory requirements, and ease of implementation. This youtube video shows the relative sorting speed (slowed down for visualization purposes) of 9 popular sorting methods.

Implement bubble sort and insertion sort algorithms and compare the speed of each of your implementations to the built-in sorting method in MATLAB. All of your sorting methods should sort in descending order. Your bubble sort and insertion sort algorithms should be home-brew; that is, they may utilize loops, decisions, indexing, concatenation, etc. but should not utilize high-level, built-in functions such as `sort()`, `issorted()`, `sortrows()`, `circshift()`, `max()`, `min()` or the like.

The bubble sort algorithm consists of stepping through each index of an incoming list (i.e. vector) of numbers, comparing adjacent terms, and swapping the terms if they are in the wrong order. This process is repeated until the list of numbers is fully sorted.

Consider a vector `v = [6.3, 5.4, 3.4, 9.7, 8.5]`. A bubble sort algorithm used to sort the vector from largest to smallest values would take the following steps. The bold/underlined indices are under consideration in each step.

step 1: `[`**`6.3, 5.4`**`, 3.4, 9.7, 8.5]` → 6.3 > 5.4, the indices are not swapped

step 2: `[6.3,` **`5.4, 3.4`**`, 9.7, 8.5]` → 5.4 > 3.4, the indices are not swapped

step 3: `[6.3, 5.4,` **`3.4, 9.7`**`, 8.5]` → 3.4 < 9.7, the indices are swapped

step 4: `[6.3, 5.4, 9.7,` **`3.4, 8.5`**`]` → 3.4 < 8.5, the indices are swapped

step 5: `[`**`6.3, 5.4`**`, 9.7, 8.5, 3.4]` → 6.3 > 5.4, the indices are not swapped

step 6: `[6.3,` **`5.4, 9.7`**`, 8.5, 3.4]` → 5.4 < 9.7, the indices are swapped

step 7: `[6.3, 9.7,` **`5.4, 8.5`**`, 3.4]` → 5.4 < 8.5, the indices are swapped

etc…

This algorithm continues until a full sweep of the vector does not yield any swaps (i.e. the list is fully sorted)!

Your bubble sort function should have the following declaration.

```
function [t,v] = bubble_sort(v)
```

Input `v` is a vector of size 1 x *n*, where *n* is an integer greater than 1, output `t` is the time it took to complete the sort of input vector `v`, and output `v` is the sorted vector.

The insertion sort algorithm consists of investigating each index of a set of numbers to be sorted (starting from the second index of the set) and determining where the index should be inserted

in the set of numbers that has already been sorted (i.e. the set of numbers preceding the number to be sorted).

Consider the same vector used to describe the bubble sort algorithm, `v = [6.3, 5.4, 3.4, 9.7, 8.5]`. An insertion sort algorithm used to sort the vector from largest to smallest values would take the following steps. The bold/underlined indices are under consideration in each step.

step 1: `[6.3, **5.4**, 3.4, 9.7, 8.5]` → index 2 is under consideration, 5.4 is smaller than the value stored in the index preceding it (index 1) and is not relocated

step 2: `[6.3, 5.4, **3.4**, 9.7, 8.5]` → index 3 is under consideration, 3.4 is smaller than the values stored in the indices preceding it (indices 1 and 2) and is not relocated

step 3: `[6.3, 5.4, 3.4, **9.7**, 8.5]` → index 4 is under consideration, 9.7 is larger than the values stored in the indices preceding it (indices 1, 2, and 3), 9.7 is placed in index 1 and all other indices are shifted to the right

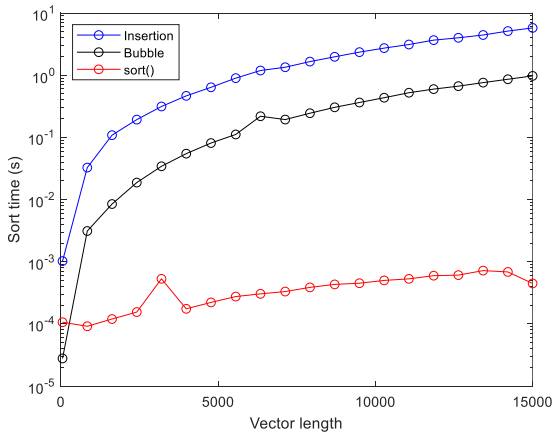
step 4: `[9.7, 6.3, 5.4, 3.4, **8.5**]` → index 5 is under consideration, 8.5 is larger than some of the values stored in the indices preceding it (indices 2, 3, and 4), 8.5 is placed in index 2, 9.7 remains in index 1, and all other indices are shifted to the right, the index is now fully sorted and the final form is `[9.7, 8.5, 6.3, 5.4, 3.4]`

Your insertion sort function should have the following declaration.

```
function [t,v] = insertion_sort(v)
```

Input `v` is a vector of size 1 x *n*, where *n* is an integer greater than 1, output `t` is the time it took to complete the sort of input vector `v`, and output `v` is the sorted vector.

Create a script that calls on your `bubble_sort()`, `insertion_sort()`, and MATLAB's built-in `sort()` function 20 times feeding in 1 x *n* vectors of uniformly distributed random numbers (use the `rand()` function) with *n* increasing approximately linearly from 50 to 15000. Produce a plot of the sort time in seconds as a function of the length of the sorted vector. Your plot should look somewhat like the following. Note that your sort times will be dependent on the efficiency of your code. You may or may not find that your insertion sort algorithm is faster than your bubble sort algorithm depending on the small nuances of your algorithmic choices.



Respond to the following questions.

i) What algorithm(s) does MATLAB's `sort()` function utilize?

ii) Which sorting method of the three you investigated would you select if sorting a large set of numbers and why?

Upload your sorting functions as `bubble_sort.m` and `insertion_sort.m`, name the script that calls your sorting code and the resulting plot reasonably, and the identify the names of all files in your README.txt. Answer the questions posed above in your README.txt.