Part 1. Sorting and searching

1) (7 pts) You have been provided with a function, *newSort*, which sorts a list of integers provided as input and returns the sorted list. Analyze the correctness and efficiency of this function. Your analysis of the efficiency should include:

a) a clear and concise discussion of the running time and how it depends on the input,

b) a critical comparison of newSort and merge sort

c) one or more figures illustrating key trends in the actual computation time required by the function, and d) a description of and explanation of the trends shown in the figure(s). The code that generates your figures should be placed in *time_newSort*.

Place your discussion and figure(s) in the appropriate section of your *tex* file. The __name__=='__main__' portion of the module should call *time_newSort* and generate any figures included in your submission.

Note: You do not need to explain the correctness or cost of the *merge* routine.

2) (3 pts) Given an N-element list of integers, *L*, with indices from 0 to N-1 (inclusive), a *trough* corresponds to an index, *i*, where all elements of *L* directly adjacent to *L[i]* are greater than or equal to *L[i]*. The list, L=[0,1,2,4,3,5]L=[0,1,2,4,3,5], has troughs at indices 0 and 4.

Complete the function, *findTrough* in *p1_dev.py*, so that it efficiently finds and returns *a* trough of the list provided as input. If *L* does not contain a trough, the function should return −(N+1). In the appropriate section of your *tex* file, add a brief description of the algorithm that you have implemented and a concise discussion of your algorithm's asymptotic running time.

Part 2. Working with DNA sequences

1) (3 pts) Consider a DNA sequence, *S*, provided as input. The goal is to convert this to a string of amino acids corresponding to the codons in *S*. The first three letters in *S* correspond to the first codon and the first amino acid in the sequence. The genetic code for the second amino acid corresponds to the 4th-6th letters and so on.

The function *CodonToAA* in *p12_dev.py* converts individual codons (three uppercase letters) into amino acids (single lowercase letter). Complete *DNAtoAA* so that it converts *S* into a string, *AA*, of all distinct amino acids contained in the string ordered as they first appear in *S*.

Note that the string should not include any amino acid more than once. For example, the string "ATAATCATAATG" should be converted to "im". Add clear and concise discussions of your algorithm and its running time to your *tex* file.

Note: You are not required to use *CodonToAA* in your code

2) (7 pts) Now, consider a list of DNA sequences provided as input. Each element of the list is an N-character string, and we are interested in finding “aligned” k-mer pairs in “adjacent” sequences. A k-mer is a sequence of k consecutive nucleotides in a DNA sequence, two sequences are adjacent if they are adjacent to each other within the list (L[i] and L[i+1] are adjacent). And a pair is aligned if each k-mer is found in the same location in the two adjacent sequences. Consider the list, [“GCAATTCGT”,”TCGTTGATC”] and the k-mer pair[(“TCG”,”GAT”)]. Then this is an aligned 3-mer pair in the two (adjacent) sequences with starting index 5. You will be provided with a list of k-mer pairs, and you will search for these pairs within the input list of sequences.

i) Complete pairSearch in *p12_dev.py* so that it efficiently finds the locations of all aligned k-mer pairs (contained in input list, *pairs*) within adjacent DNA sequences (contained in input list, *L*). Each element of *pairs* is a tuple consisting of two strings. The function should return a list of 3-element sublists. Each sublist should contain: 1) the starting location within the sequences for the pattern that has been found, 2) the index corresponding to the first of the two sequences in which the pattern was found, and 3) the index in *pairs* corresponding to the pattern that was found. So if the the pair corresponding to pairs[4] is found in L[3][5:9] and L[4][5:9], the output list should contain the sublist, [5,3,4]. The order of the sublists within the output list is not important. If no aligned pairs are found, return an empty list. You should use the Rabin-Karp algorithm modified as needed for this specific problem. You may assume that there is no performance penalty associated with calculations involving large integers.

ii) Provide a concise description of your code along with a careful analysis of its running time in your *tex* file. Your analysis should critically compare your method with the naive search algorithm from lecture.

We are generally interested in cases where N is very large though you may assume that *L* will comfortably fit in your computer’s main memory. You may also assume that N≫kN≫k and len(L)≫klen(L)≫k.