1c) Each point was averaged over 20 runs.

Out of all the algorithms, algorithm 1bii has the fastest insertion time, shortly followed by algorithm 1a). The reason why algorithm 1a is so fast is because it's universal – it's purely O(n) as it goes through the array of the string, and it builds a second array as it goes, so no modification of any data structure is performed I'll focus on the 1b structures from now on.

Loading the string for 1bi is significantly faster than for 1bii – that is because calculating the hashes, them being as big as they are, takes significantly more time. I took a random high(10^5 degree) prime number to minimize the hash collisions at the cost of loading time. This was a right decision – insertion time remains constant for every insertion strand(10, 100, 1000) – at worst 3.5 sec. The constant α (denoted as alphabetLength in the code) inside the hashing function for strands 10 and 100 was 4, and for 1000 was 2 – this is because otherwise the numbers would get too big for the machine to handle (which could be remedied by a smaller prime number, but I preferred this solution). Interestingly, this doesn't seem to have influenced runtime at all.

What is more, the loading time for 1bi remains constant regardless of the length of the splicer DNA strand. That is expected behavior as, contrary to the 1bii attempt, the building of the structure is not dependent on the splicer (while hashes are calculated based on the length of the splicer).

Overall, interestingly, 1bii is the slowest algorithm – while it takes half the time to insert and double the time to load, loading time is usually a factor of 10 bigger than insertion time. When judging the insertion time only, the time to run the algorithm 1bi will be O(n*m), where m is the length of the target string (as in each iteration of the algorithm we build the string we check) of O(n*m), while the hash – based approach will be – on average – of O(n+m) complexity.

A note about runtime – please note that generating a string of lengths 4*10^7 takes 40 seconds, loading into a structure will take up to 60 seconds(for hash based), and then using the algorithm will be very quick (3-11 sec). Most of the time testing is actually spent generating a random viable string (with the insertion strand appearing at least once).

2a)

```
findSequence(string1, string2, lengthOfString, passToNext = 0):
        #base case – return an empty string
        If lengthOfString == passToNext:
                Return " "
        substrings1 = [all substrings of string1 of length (lengthOfString – passToNext)]
        substrings2 = [all substrings of string2 of length (lengthOfString – passToNext)]
        commonSet = substrings1 && substrings2          #conjunction on set will find common elements
        if length(commonSet)>0:
                Return commonSet[0]                     #return the first common substring found
        #if after the loop no match has been found, call the function recursively
```

```
        passToNext +=1
        Return findSequence(string1, string2, lengthOfString, passToNext)
```

When trying to run this algorithm on a string length higher than 1000, stack overflow (even when setting a high recursion call limit) happened. Initially, my function for finding the substrings of a given length was as follows:

```
def findSubstrings(string, lengthOfSubstring, position = 0, substringList = []):
  if lengthOfSubstring + position == len(string) + 1:
    return substringList
  else:
    substringList.append(string[position:lengthOfSubstring + position])
    findSubstrings(string, lengthOfSubstring, position+1, substringList)
    return substringList
```

Here, the recursive approach didn't work because of what I mentioned above and so I modified this approach into an iterative one. That was a solution only until string length of 2300.

In O notation, let's assume worst case scenario – the common string will be of length 0 (no common substring found)
findSequence will be called lengthOfString times (n) – sum from i = 0 to n

Finding substrings: O(2*i) (where i is an element from set (1..n)) = 2i

Turning substring arrays into sets: i

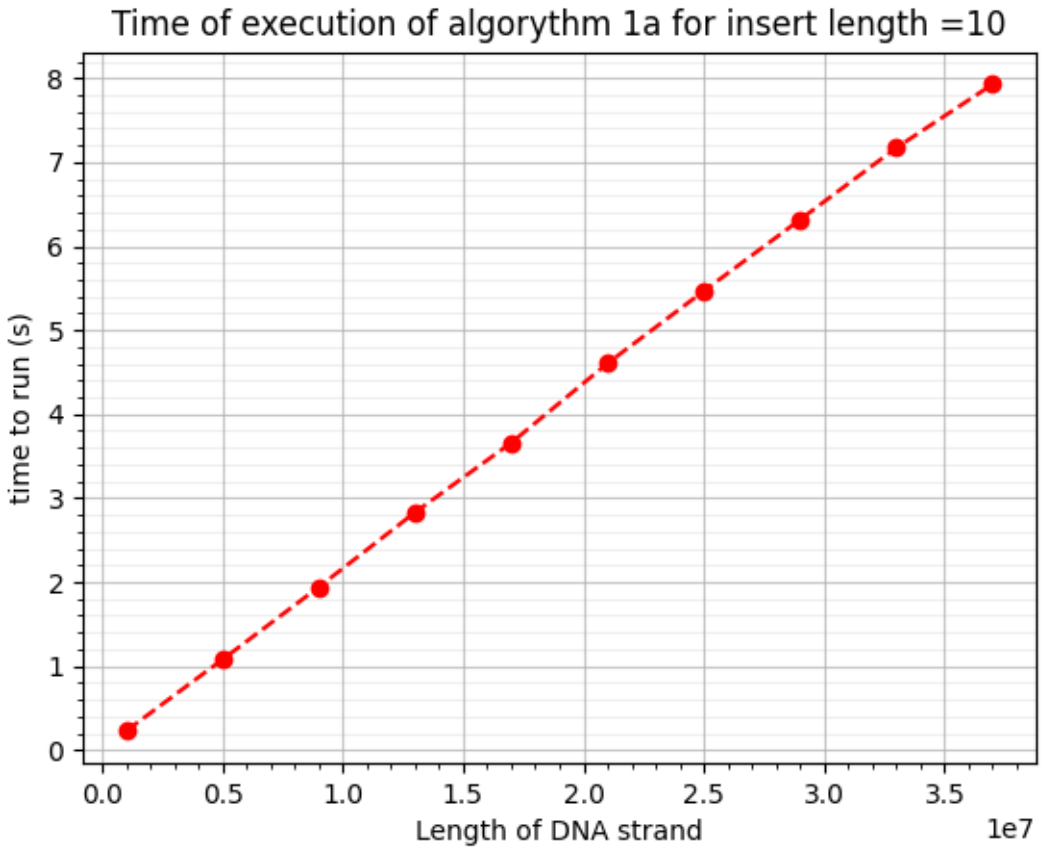
Set intersection (according to python wiki): min(len(set1),len(set2)) – i

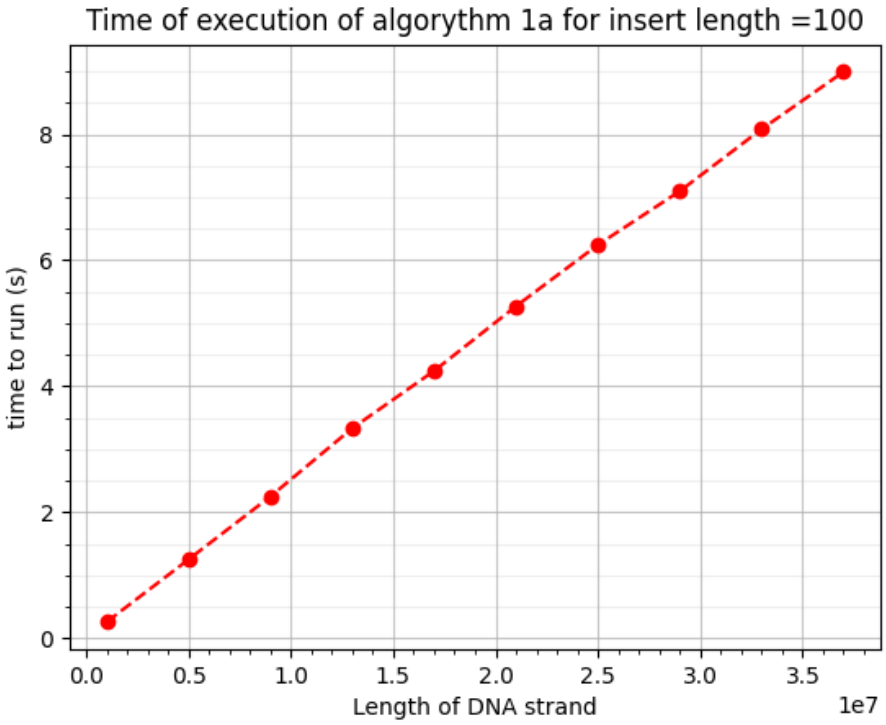
In total: $\sum_{i=0}^{n}\left(\sum_{i=1}^{n} 4\,i\right) = 2\,n\,(n+1)^2$ **That is O(n^3) complexity.**

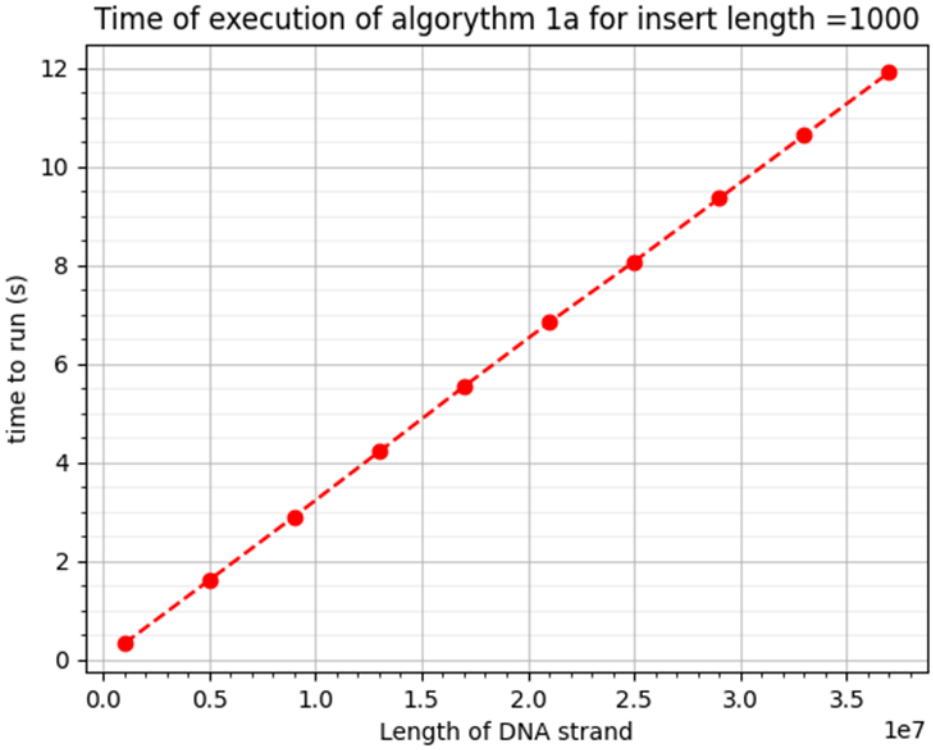
2c)

Comparing the two graphs, it can be observed that the recursive algorithm, while it starts off running roughly as fast as the dynamic approach, it's runtime quickly starts climbing – in fact, it starts taking up so much stack memory (because of the piling recursive calls), it will quickly overflow when reaching string lengths above 2300. Assuming we have memory available, it's running time would become significantly slower than that of dynamic programming's. While the dynamic programming's solution creates a table of size (n^2), it still takes less space than the recursive solution; The reason for that is even for strings of length 2000, the longest common substring will be on average 7-10 characters long – so the recursive function on its bottom level still has to keep ~2000 pointers to get back to the surface! What is more, dynamic programming has superior time complexity O(n^2) compared to O(n^3) (where m, on average, will be some fraction of n).
The only reason I could think of why anyone would prefer to use the recursive approach is when it is known that there is a very long common substring (say n/2) – as then the recursive approach will be faster, as the dynamic one does not take into account the length of the substring.

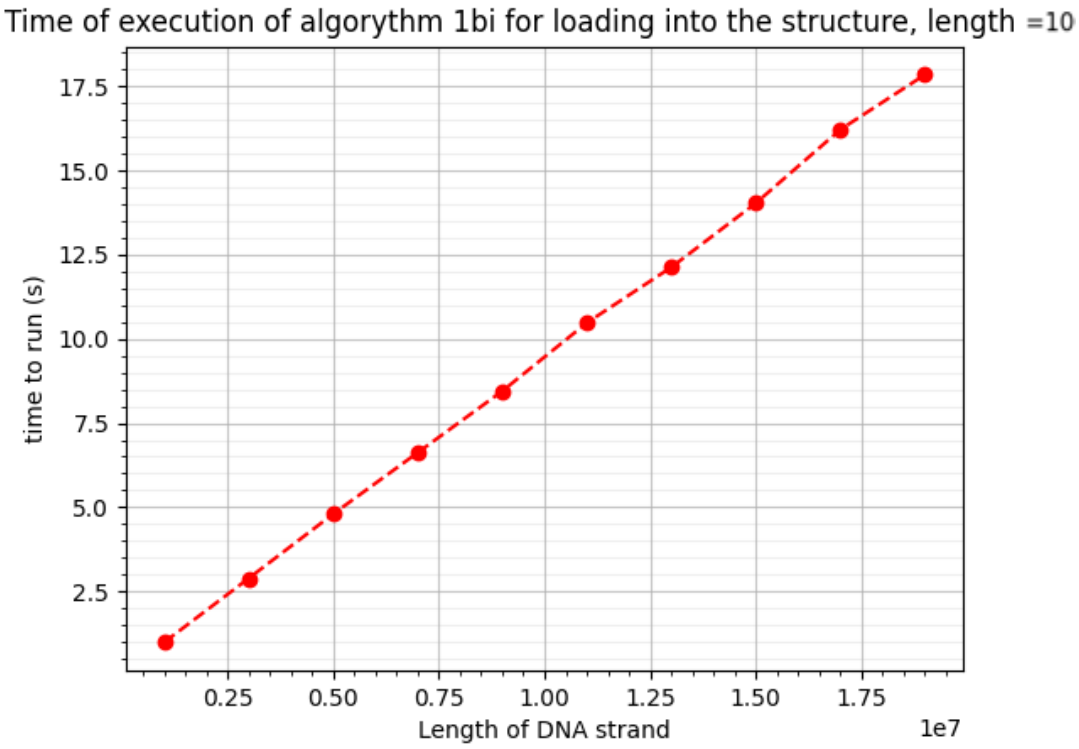
1c) Algorythm 1a)

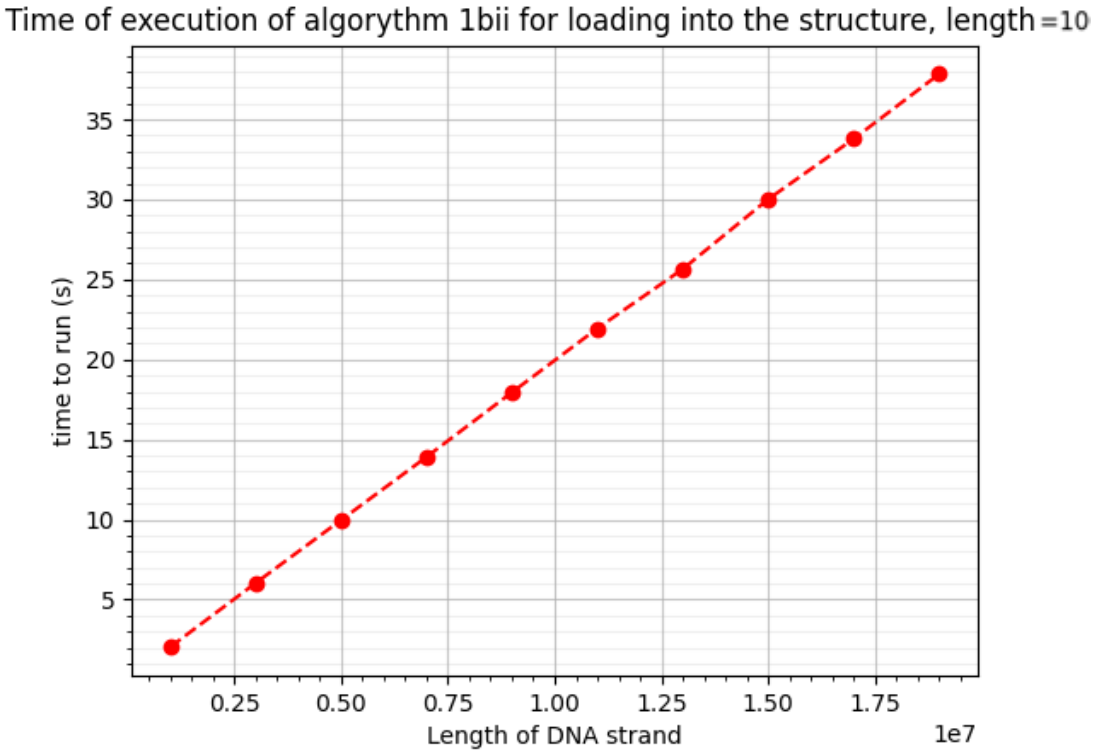


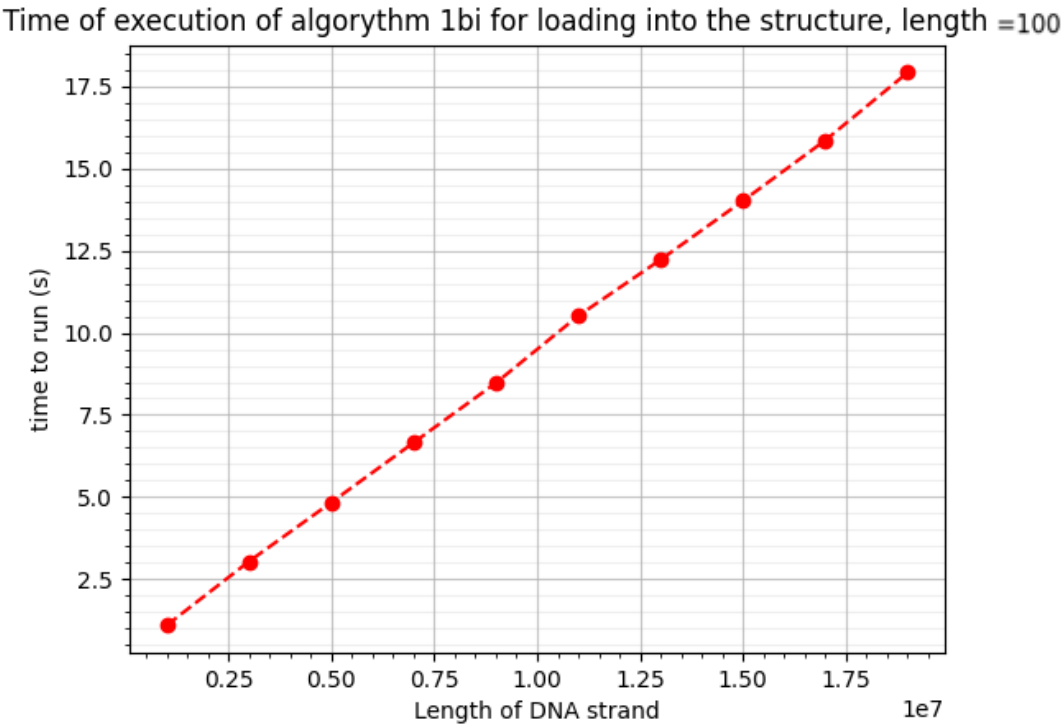

Time of execution of algorythm 1a for insert length =100
time to run (s)
0
2
4
6
8
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
Length of DNA strand
1e7

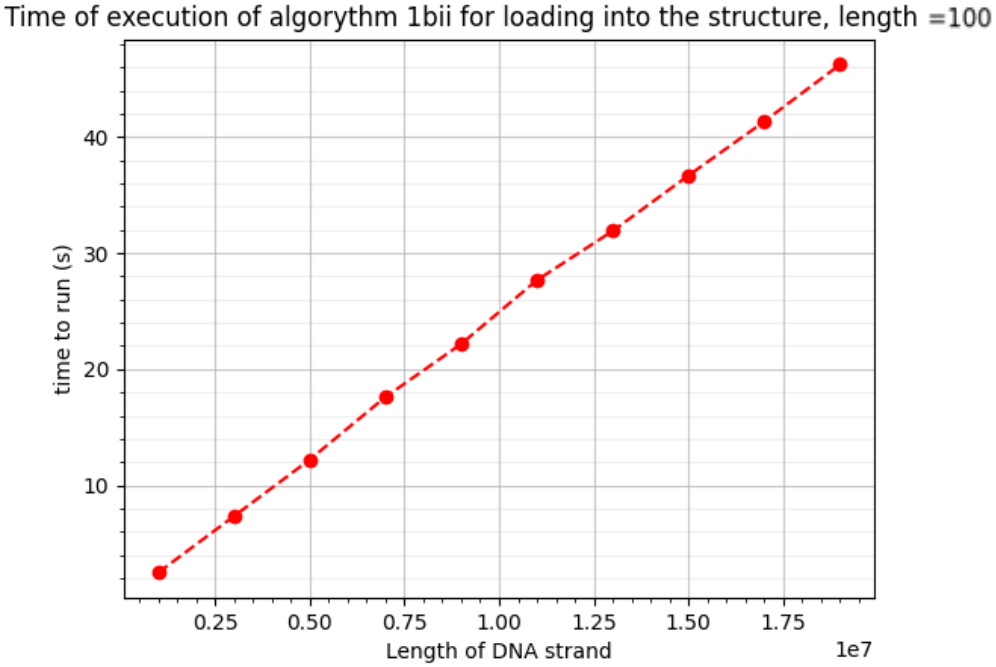

Time of execution of algorythm 1a for insert length =1000
time to run (s)
0
2
4
6
8
10
12
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
Length of DNA strand
1e7

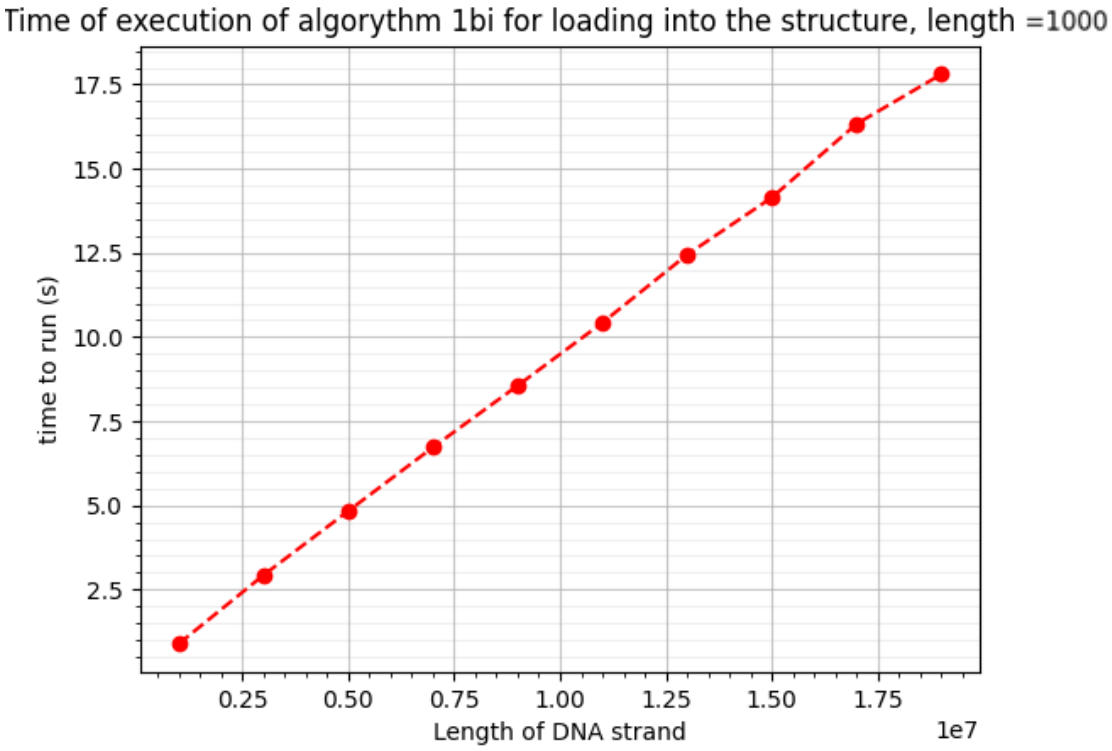
Algorythm 1bi vs 1bii: load times

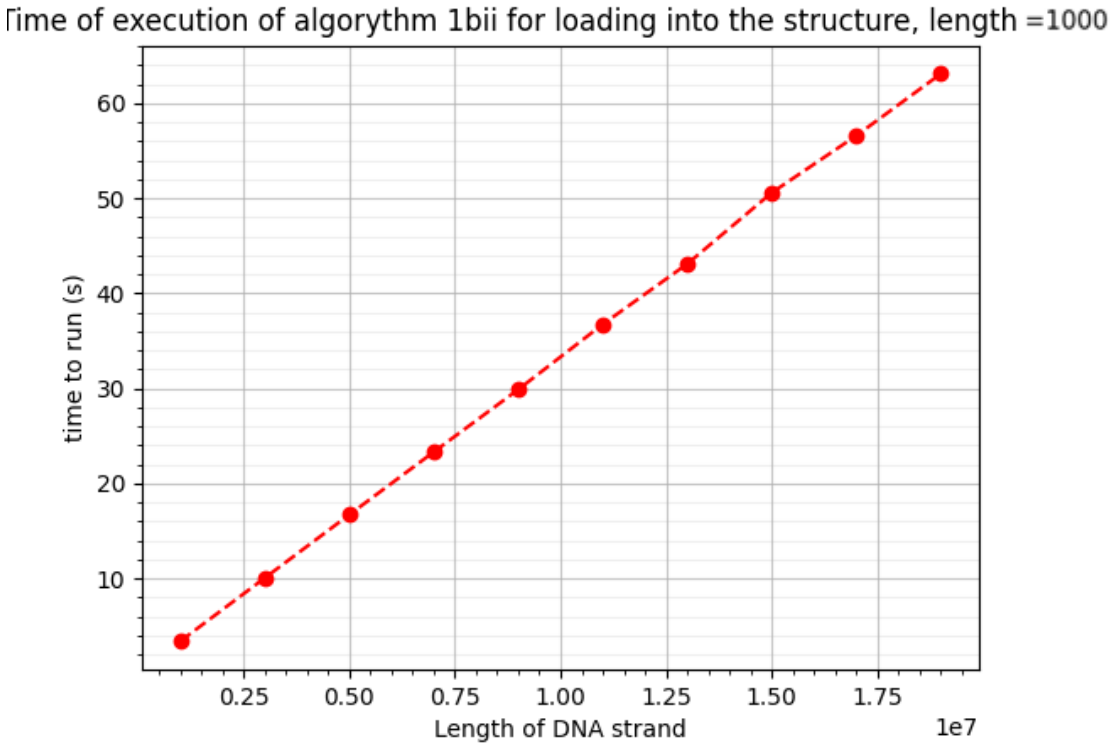




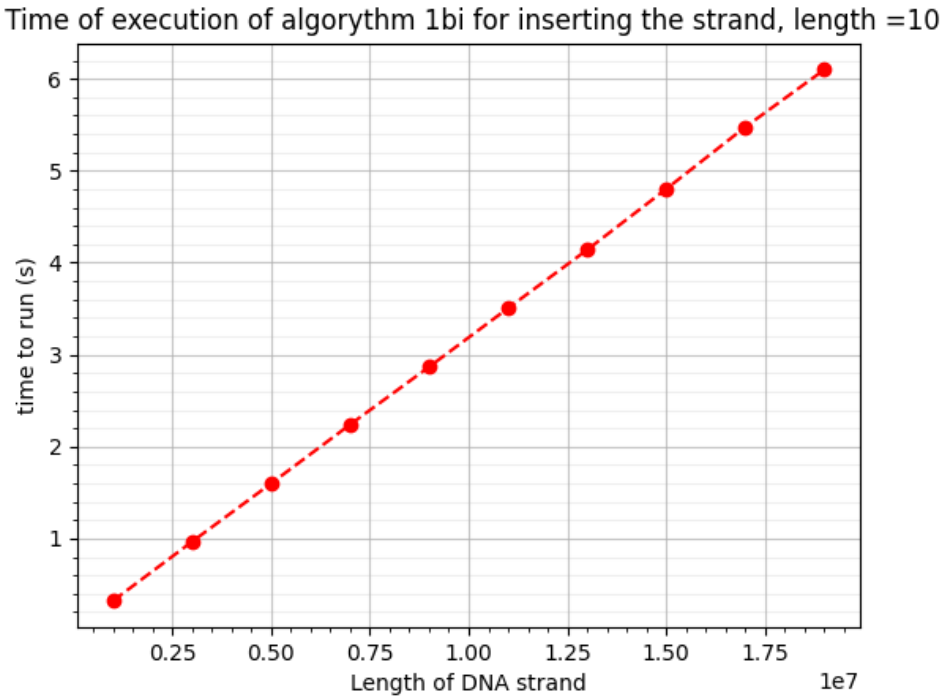

Time of execution of algorythm 1bi for loading into the structure, length =100
time to run (s)
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.25
0.50
0.75
1.00
1.25
1.50
1.75
Length of DNA strand
1e7

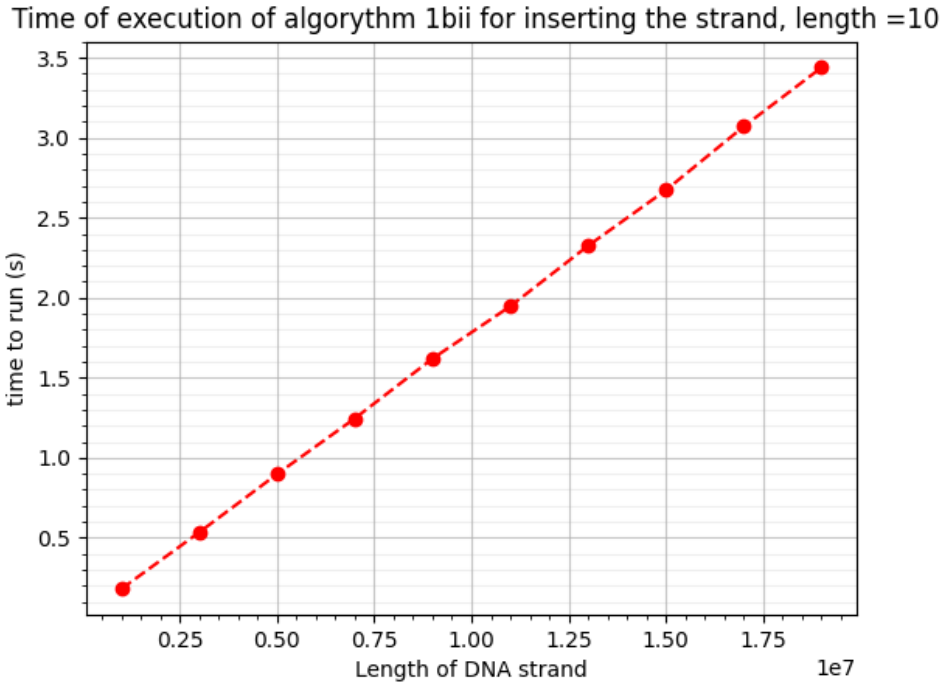

Time of execution of algorythm 1bii for loading into the structure, length =100
time to run (s)
40
30
20
10
0.25
0.50
0.75
1.00
1.25
1.50
1.75
Length of DNA strand
1e7

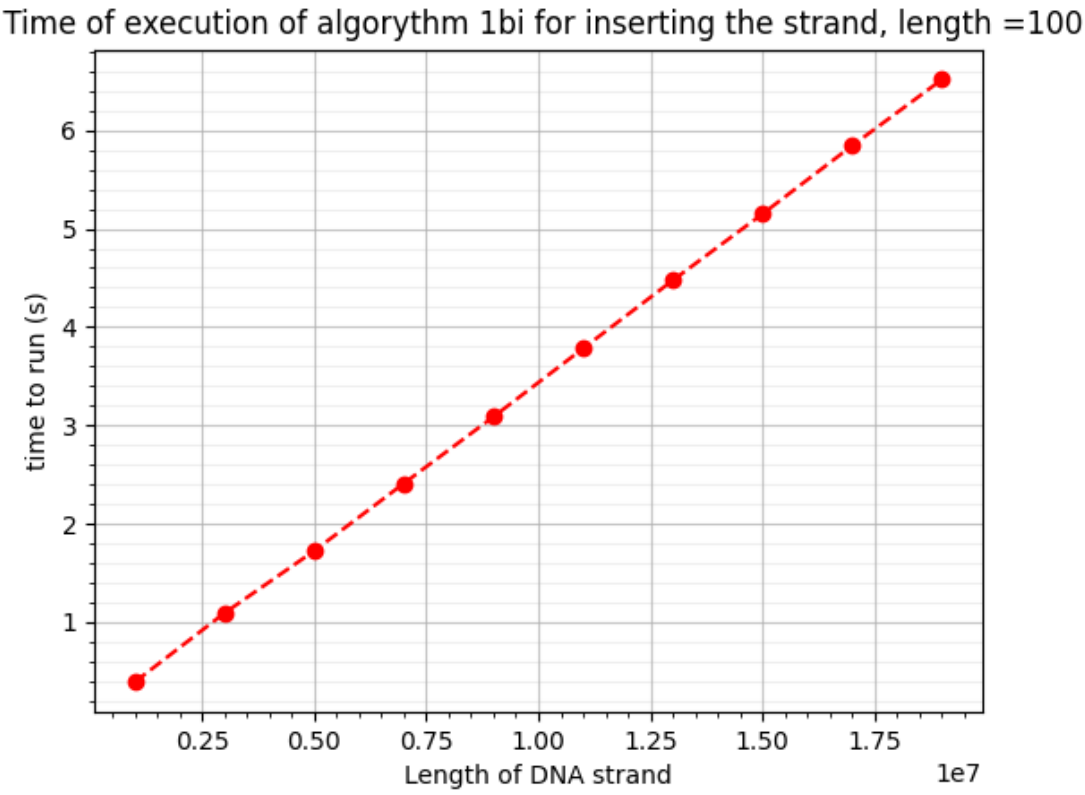

Time of execution of algorythm 1bi for loading into the structure, length =1000
time to run (s)
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.25
0.50
0.75
1.00
1.25
1.50
1.75
Length of DNA strand
1e7

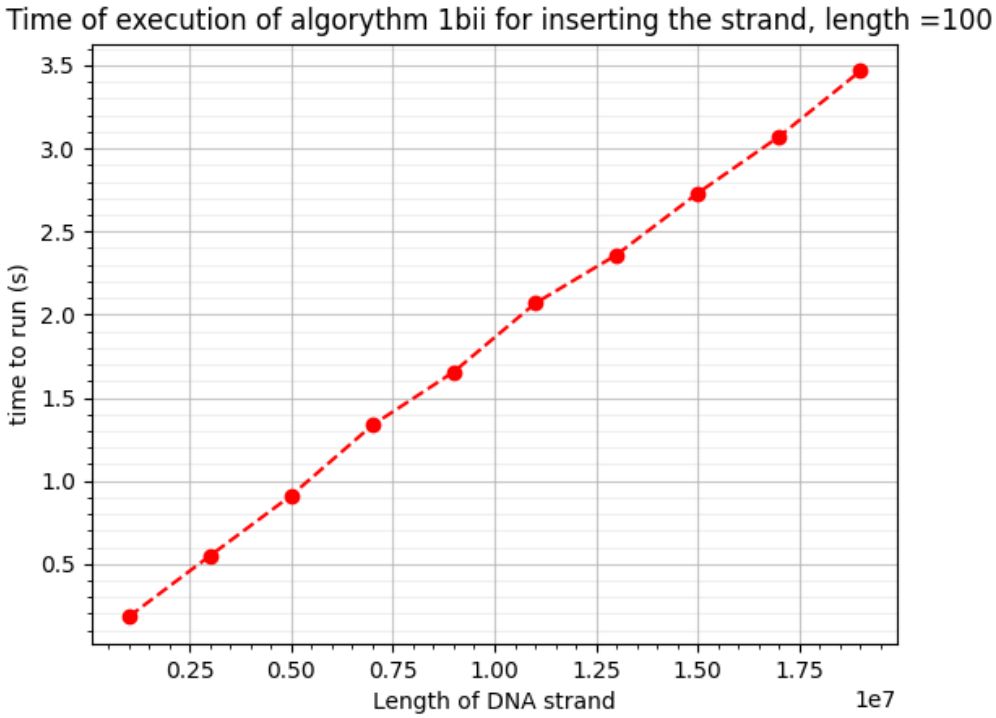

Time of execution of algorythm 1bii for loading into the structure, length =1000
time to run (s)
60
50
40
30
20
10
0.25
0.50
0.75
1.00
1.25
1.50
1.75
Length of DNA strand
1e7

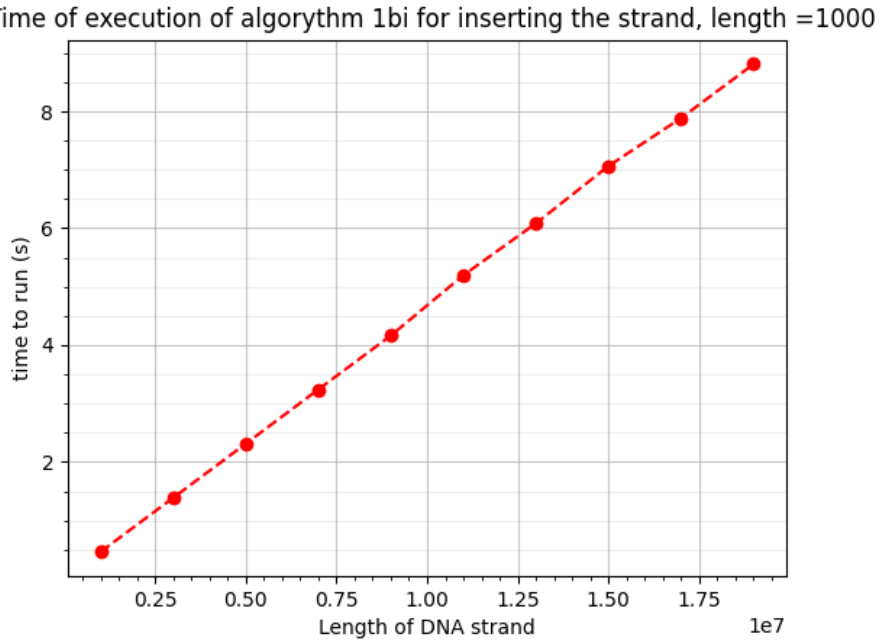
Algorythm 1bi vs 1bii, insert times

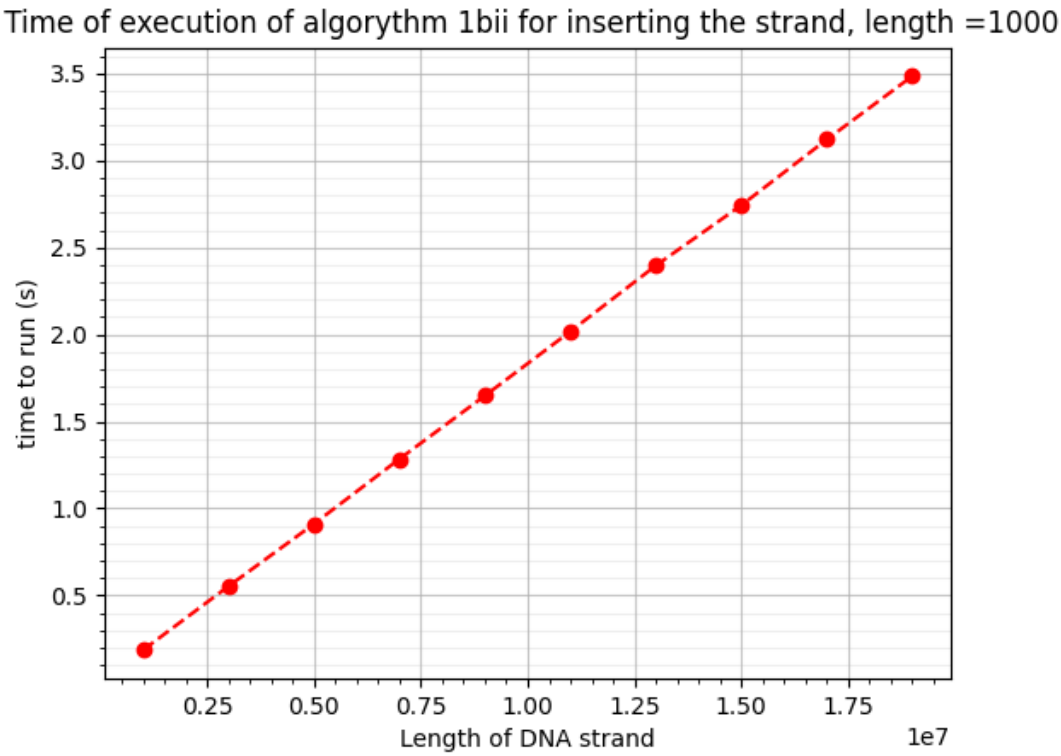





Time of execution of algorythm 1bi for inserting the strand, length =100
time to run (s)
1
2
3
4
5
6
0.25
0.50
0.75
1.00
1.25
1.50
1.75
Length of DNA strand
1e7


Time of execution of algorythm 1bii for inserting the strand, length =100
time to run (s)
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0.25
0.50
0.75
1.00
1.25
1.50
1.75
Length of DNA strand
1e7


Time of execution of algorythm 1bi for inserting the strand, length =1000
time to run (s)
2
4
6
8
0.25
0.50
0.75
1.00
1.25
1.50
1.75
Length of DNA strand
1e7


Time of execution of algorythm 1bii for inserting the strand, length =1000
time to run (s)
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0.25
0.50
0.75
1.00
1.25
1.50
1.75
Length of DNA strand
1e7

Algorythm 2a vs 2c: execution time