Overview

Next Generation Sequencing (NGS), a sequencing approach which works by breaking up the DNA into short fragments, replicating them in large quantities, then sequencing those short fragments in a high-throughput fashion. The output of an NGS sequencer is a large collection of such disjointed sequences, called "reads". For the purpose of this puzzle, each read has two properties: (1) a start position within the genome and (2) its length. Both properties are expressed in base pair units, i.e. letters of DNA like A,T,C,G. For example, the read "AAATCGA" has length 7.

A key component of making sense of NGS data is calculating "coverage" (also known as "read depth") at a given genomic position. At its most basic, coverage is simply the number of reads overlapping a position in the genome. High coverage gives a measure of confidence in the sequencing results.

There are two csv files:

reads.csv : this file contains approximately 250 reads, one read per row, with two columns corresponding to the start position and length of the read. For example, the first read starts at position 101843359 and has a length of 151 base pairs.

loci.csv : this file contains 30 positions of interest and a blank "coverage" column. In the course of this exercise, you will populate the coverage column with the number of reads overlapping that position.

This code calculates the coverage for each of the positions in loci.csv, based on the reads in reads.csv.

Data Structure: Nested SortedDictionary (or TreeMap)

SortedDictionary is a generic collection which is used to store the key/value pairs in the sorted form and the sorting is done on the key. It is dynamic in nature means the size of the sorted dictionary is growing according to the need. It provides fastest insertion and removal operations for unsorted data. SortedDictionary is based on Binary Search Tree. As a result (as illustrated in Table 1.), operations of both insertion and removal take O(log n).

Table 1. Time complexities of important operations in the Dictionary and SortedDictionary

Operation	**Dictionary**	**SortedDictionary**
Get[key]	O(1)	O(log n)
Add(key,value)	O(1) or O(n)	O(log n)
Remove(key)	O(1)	O(log n)
ContainsKey(key)	O(1)	O(log n)
ContainsValue(value)	O(n)	O(n)

I used the nested SortedDictionary data structure to solve your problem because it is automatically sort the keys and values in O(log(n)).

1- My script reads the *reads.csv* file line by line and the *position* are inserted as keys. For *coverage*, again another SortedDictionary is considered where *coverage* is considered as a key for this nested SortedDictionary (see Table 2.).

Table 2. Data structure schema with m unique positions

Key	**Value**

$position_1$	Key	value
	$coverage_1$	Number of times $coverage_1$ appears for $position_1$
	.	
	.	
	.	
	$coverage_j$	Number of times $coverage_j$ appears for $position_1$

.
.
.

$position_m$	Key	value
	$coverage_1$	Number of times $coverage_1$ appears for $position_1$
	.	
	.	
	.	
	$coverage_j$	Number of times $coverage_j$ appears for $position_1$

For example for following positions and coverages, the nested SortedDictionary is initialized as follows:

position	coverage
3	0
4	5
2	0
3	2
4	5
1	2
3	0
4	5
4	1
2	2

→

key (position)	**value** ({key (coverage): value(number of observations)})
1	{2:1}
2	{0:1, 2:1}
3	{0:2, 2:1}
4	{1:1, 5:3}

Or {**1**:{2:1}, **2**:{ 0:1, 2:1}, **3**:{0:2, 2:1}, **4**:{1:1, 5:3}}

Figure 1. Example of inserting reads.csv file into the nested SortedDictionary.

```
def read_csv_file(read_file_path):
    #data structure nested SortedDictionary
    treeMap = SortedDict()

    #read from file
    with open(read_file_path) as file:
        #ignore first line of file
        next(file)

        for line in file:
            #this variable indicates the number of 'identical values' for a 'key'
            count = 0

            new_key, new_value = (int(x) for x in line.split(','))


            if new_key in treeMap.keys(): #O(1) because of the internal hash function for SortedDict
                index = 0
                threshold = len(treeMap[new_key])
                for value, count in treeMap[new_key].items(): # It would be around O(1) since for each key usually (maybe by the way!) there
                    if new_value == value:
                        treeMap[new_key].update({new_value:(count + 1)}) #O(1)
                        break

                    index = index + 1
                    if index == threshold:
                        treeMap[new_key].update({new_value:1}) #O(1)
                        break


            else:
                treeMap.update({new_key:SortedDict({new_value:1})}) #O(1)

    return treeMap
```

Figure 2. Script of inserting reads.csv file into nested SortedDictionary

2- After inserting all values of *reads.csv* file, my script reads *loci.csv* file

3- For each *position* in *loci.csv*, we find the last index of *position* in our data structure which the *position* (key of data structure) is less than or equal to the *position* in loci.csv. For above example{**1**:{2:1}, **2**:{0:1, 2:1}, **3**:{0:2, 2:1}, **4**:{1:1, 5:3}}, assuming the *position* in *loci.csv* file is **3** then the last index is 2 (in other words **1**, **2**, **3** ≤ to **3**).

```
def binary_search_key(new_key, sorted_Dict):

    first_key = 0
    last_key = len(sorted_Dict)-1
    found = False

    if sorted_Dict.keys()[first_key] == new_key:
        found = False
        mid_key = first_key
    if  sorted_Dict.keys()[last_key] == new_key :
        found = False
        mid_key = last_key

    while (first_key<=last_key and not found):
        mid_key = (first_key + last_key)//2
        if sorted_Dict.keys()[mid_key] == new_key :
            return mid_key
        else:
            if new_key < sorted_Dict.keys()[mid_key]:
                last_key = mid_key - 1
                if last_key < 0:
                    return -1

            else:
                first_key = mid_key + 1
                if len(sorted_Dict) <= first_key:
                    return len(sorted_Dict) - 1

    if sorted_Dict.keys()[mid_key] <= new_key:
        return mid_key
    else:
        return mid_key - 1
```

Figure 3. Script binary search on positions

4- We do another binary search for *coverage* of *positions* to find the first index of search in nested SortedDictionary that the value of *coverage* - *positions* is bigger than or equal to the *coverage*. For example for {**1**:{$_2$:^{1}}, **2**:{ $_0$:^{1}, $_2$:^{1}}, **3**:{$_0$:^{2}, $_2$:^{1}}, 4:{$_1$:^{1}, $_5$:^{3}}} and the value of 3 in *loci.csv*, the above binary search told us that the range of desired keys is 0 to 2 (or **1, 2, 3**):

   a. For **1**:{$_2$:^{1}}, the second binary search finds keys (coverage) that is equal to or bigger than 2 (= **3**-**1**). Therefor the first index is 0 ($_2$≤2) and the desired range indexes are 0 to 0. So for position **1**, coverage $_2$, we have ^{1} observation. In other words, 1 coverage is here.

   b. For **2**:{ $_0$:^{1}, $_2$:^{1}}, the value of **3** – **2** is 1, then the first index that (its value ≤1) is index 1 (or $_2$) and the desired range indexes are 1 to 1, since we have just ^{1} observation for $_2$, then there is 1 coverage here.

c. For **3**:{$_0$:^{2}, $_2$:^{1}}, (**3** –**3**) = 0, then the first index that (its value≤0) is 0. The desired range indexes are 0 to 1 (or $_0$ and $_2$) and their observation are ^{2} and ^{1}, therefore we have 3 (^{2}+^{1}) coverages here.

d. The number of coverages for **3** in {**1**:{$_2$:^{1}}, **2**:{ $_0$:^{1}, $_2$:^{1}}, **3**:{ $_1$:^{2}, $_3$:^{1}}, **4**:{$_1$:^{1}, $_2$:^{3}}} is ^{1}+^{1}+^{2}+^{1}=5

```
def binary_search_value(new_value, sorted_Dict):

    first_value = 0
    last_value = len(sorted_Dict)-1
    found = False

    if sorted_Dict.keys()[first_value] == new_value:
        found = False
        mid_value = first_value
    if  sorted_Dict.keys()[last_value] == new_value :
        found = False
        mid_value = last_value

    while (first_value<=last_value and not found):
        mid_value = (first_value + last_value)//2
        if sorted_Dict.keys()[mid_value] == new_value :
            return mid_value
        else:
            if new_value < sorted_Dict.keys()[mid_value]:
                last_value = mid_value - 1
                if last_value < 0:
                    return 0

            else:
                first_value = mid_value + 1
                if len(sorted_Dict) <= first_value:
                    return len(sorted_Dict)

    if sorted_Dict.keys()[mid_value] >= new_value:
        return mid_value
    else:
        return mid_value + 1
```

Figure 4. Script of binary search on coverages

Time complexity:

- Time complexity for inserting **n** positions and coverages of *reads.csv* into nested SortedDictionary, by assuming m positions are unique and each of m unique positions, has m_i coverages (m_1 to m_m).

Time complexity of inserting first position in nested SortedDictionary, is O(log(1)) and also inserting m_1 coverages of first unique position takes O(log(m_1)). Time complexity of inserting second position in nested SortedDictionary, is O(log(2)) and inserting coverages of second position takes O(log(m_2)). Time complexity of inserting m^{th} position in nested SortedDictionary, is O(log(m)) and inserting its m_m coverages takes O(log(m_m))

We can consider log(1) + log(2) + …+ log(n) = O(logn!) as n*(long(n)).

Then time complexity for inserting n position from *reads.csv* file into our data structure is: $n * log(n) + m * log(m)$). Since m≤n then time complexity is O(n*log(n))

For boundary conditions:

- **Worst case scenario**→ If there is just one unique number (1 position value that has n unique coverage values) then (in other words, m=1, m_1=n)

For n positions (that are the same as each other therefore we have one key!), it is n*log(1)=0, for n values of that key, it is log(1) + log(2) + …+ log(n) = log(n!), or (n*log(n))

Therefore for worst case scenario: O(n*log(n))

- **Best case scenario**→ If all position are the same as each other and all coverages are the same as each other too:

For n positions (that are the same as each other therefore we have one key!), it is n*log(1)=0, for n values of that key that are the same as each other it is n*log(1)=0

Therefore for best case scenario: O(1)

- Time complexity for writing and reading into/of file is O(n).
- If we have a **n** positions in *loci.csv* and we have **n** positions in *reads.csv* file, the time complexity for our two binary searches are (to search n positions of *loci.csv* in data structure of **n** positions of *reads.csv*) around **n*log(n)**
- **Time complexity of this script is n*(log(n)) for reading from file, writing in file, inserting in data structure, searching in data structure of all positions of loci.csv file and all positions and coverages of reads.csv file.**