

Question 3:

José Murta 21032320

My pair is: ArrayList and LinkedList.

Starting off with the method that is faster in the class ArrayList than it is on LinkedList. The method I choose is: `get(int index)`. This method is faster on ArrayList because as the name of the class suggests, an ArrayList uses an array for internal storage, so when trying to access an element, with the index given, it gets you directly to that element, no matter the size of the array. The O-notation of this method is $O(1)$. However, on a LinkedList, since it consists on a chain of nodes, you cannot directly access an element. To access an element with the index given, you have to transverse forward from the beginning **(or backward from the end)**, calling next **(or previous)**, passing through every node, until you get to the element with the given index. For example, if you have a list of 100 elements, and you want to access the element in the position 50, you would have to pass through the first 50 elements, to reach the one you wanted. The O-notation of this method in the LinkedList class is, in the worst case, $O(n)$, with n being the size of the list.

On a real world circumstance, using an ArrayList can be really appropriate when you have a system that displays certain transactions and you have an ordered external database with all the transactions. If you want to access a certain transaction with a specific timestamp that is present in the database, to then display it to the user, using the `get` method from the ArrayList class on the database to get the specific element, would be the ideal, because getting any element with any index takes constant time, so the O-notation would always be $O(1)$.

Moving now to the method that is faster in the class LinkedList than it is on ArrayList, I choose the method: `add(int index, E element)`. Adding is fast on LinkedList because, if you are adding to the first position of the list, all you have to do is update the next pointer **(and previous)** and the root of the list, and if you are adding in any other position, you have to transverse to that position, and then update the next pointer of the new element, and next pointer of the old node if there was a node before adding in that position **(and the respective previous pointers)**. Adding in the beginning of the list has a O-notation of $O(1)$, and anywhere else in the worst case $O(n)$. However, this method is still faster in the LinkedList than it is on the ArrayList because, on the ArrayList, when adding at the start or middle, all the later elements of the list have to be copied forward, and this has always an O-notation of $O(n)$. For example, if you have a list of 100 elements, and you want to add an element to the position 3, if you are using a LinkedList all you have to do is transverse to the position 3 and update the next pointer of the old node that was on index 3 and the next pointer of the new node that will be on that position **(and previous pointers)**, but if you are using an ArrayList, although you can access directly to the position 3 to add the new element there, then you have to copy 97 elements forward and this operation takes a lot of time. **(if the LinkedList is implemented with a Doubly-LinkedList, adding at the start or at the end of the list is always $O(1)$, and in the middle it is $O(n)$, so in this case the `add(int index, E element)` method would even be faster than if it is implemented with a Single-LinkedList)**

On a real world circumstance, using a LinkedList to add elements can be really useful for a Mobile App that finds people that are near to you, limited to a certain radius of search. This

system could consist in a list that every time a new user gets in the radius, it would be added to the first position of the list, and when that user leaves the accepted radius, it would be removed from the list. Adding would take constant time, so $O(1)$, and removing, the more recent the adding was, the faster would it be, because there would be less elements to transverse through.

*The words in bold and involved in parentheses only apply if using a Doubly-LinkedList as implementation of the LinkedList.

6.1 Screen dump of profiler with original code

Method	Time (ms)	Percentage (%)
main	27.244	100%
genethoc.StudentSimulation.main (String[])	27.244	100%
genethoc.GenHocDNADatabase.everythingWhichShouldBeImproved ()	27.243	100%
genethoc.GenHocDNADatabase.testLookingForCrimeSceneCodes (int)	9.017	33,1%
genethoc.GenHocDNADatabase.howManySequencesForCrimeScene (String)	9.011	33,1%
genethoc.DNASequence.linkedToCrimeScene (String)	8.365	30,7%
Self time	646	(2,4%)
Self time	3,25	(0%)
genethoc.GenHocDNADatabase.getSomeRandomCrimeScenes (int)	2,53	(0%)
genethoc.GenHocDNADatabase.testLookingForFragement (int)	7.813	(28,7%)
genethoc.GenHocDNADatabase.testLookingForFragementPositive (int)	7.813	(28,7%)
genethoc.GenHocDNADatabase.searchForDNAFragment (String)	7.807	(28,7%)
genethoc.DNASequence.containsFragement (String)	7.113	(26,1%)
Self time	693	(2,5%)
Self time	6,59	(0%)
Self time	0,051	(0%)
genethoc.GenHocDNADatabase.testRemoveDuplicateDNASequences (int)	5.935	(21,8%)
genethoc.GenHocDNADatabase.removeDuplicateDNASequences ()	4.340	(15,9%)
Self time	1.543	(5,7%)
genethoc.DNASequence.sameDNASequence (String)	1.537	(5,6%)
genethoc.DNASequence.addCrimeScenseFrom (genethoc.DNASequence)	1.259	(4,6%)
genethoc.GenHocDNADatabase.makeDNASequence (String)	1.568	(5,8%)
genethoc.DNASequence.addRandomCrimeLocations (genethoc.DNASequence, java.util.Random)	1.039	(3,8%)
genethoc.GenHocDNADatabase.makeRandomDNASequence ()	527	(1,9%)
Self time	1,25	(0%)
Self time	25,4	(0,1%)
genethoc.GenHocDNADatabase.pickRandomSequence (java.util.Random)	1,25	(0%)
genethoc.GenHocDNADatabase.addDNASequence (genethoc.DNASequence)	0,172	(0%)
genethoc.GenHocDNADatabase.selfTest ()	3.122	(11,5%)
genethoc.GenHocDNADatabase.readyForSelfTest ()	2.051	(7,5%)
genethoc.GenHocDNADatabase.prepareDataBase ()	2.051	(7,5%)
genethoc.GenHocDNADatabase.contactDatabase ()	2.051	(7,5%)
Self time	0,046	(0%)
Self time	0,048	(0%)
genethoc.GenHocDNADatabase.testDatabaseIntegrity ()	1.071	(3,9%)
Self time	0,045	(0%)

6.2 Top 5 methods

1. linkedToCrimeScene(String whereID)
2. containsFragement(String fragment)
3. removeDuplicateDNASequences()
4. selfTest() -> -> contactDatabase() -> loadDNAFromDatabase()-> loadSimulatedDNAFromDatabase()
5. addRandomCrimeLocations(DNASequence it , Random rn, int maxLocations)

6.3 Put down your top 5 methods to change and why – targets

- `DNASquence.linkedToCrimeScene(String whereID)` : This method has an O-notation of $O(n^2)$ and it is called by the `GenHocDNADatabase`.
`howManySequencesForCrimeScene (String crimeSceneCode)`, making this method $O(n^3)$. The first method can be changed to take constant time $O(1)$, changing the second one to $O(n)$.
- `DNASquence.addCrimeScene (String whereID)`: This method's O-notation is $O(n)$, and it can be changed to $O(1)$. This method is called by `DNASquence`.
`addRandomCrimeLocations (DNASquence it, Random rn, int maxLocations)` that it is called to by two important methods `GenHocDNADatabase.makeDNASquence (String dna)` and `GenHocDNADatabase.removeDuplicateDNASequences()`.
- `GenHocDNADatabase.searchForDNA (String dna)`: This method has an O-notation of $O(n^2)$ because it iterates through every `DNASquence` in the database and checks if the `String dnaSequence` is the same as the one passed as parameter. It can be changed to an O-notation of $O(1)$.
- `GenHocDNADatabase.searchForDNAFragment (String fragment)`: This method's O-notation is $O(n^2)$ because it iterates through every `DNASquence` in the database and checks if the `String dnaSequence` contains the `String fragment` passed as parameter and if that is the case the `DNASquence` will be added to a list that is then returned. This method will only be called by the method `GenHocDNADatabase.testLookingForFragmentPositive`, and it is used to check if the size of the returned list is equals to zero. This can be done by a simple Boolean that is true when the fragment matches with the one `DNASquence` in the database.
- `GenHocDNADatabase.removeDuplicateDNASequences()` – This method's O-notation is $O(n^4)$, and can be change to $O(n^2)$ with the right data structures to handle the duplicates.

6.4 Big O Notation for targets

- `DNASquence.linkedToCrimeScene(String whereID)`: $O(n^2)$
- `DNASquence.addCrimeScene (String whereID)`: $O(n)$
- `GenHocDNADatabase.searchForDNA (String dna)`: $O(n^2)$
- `GenHocDNADatabase.searchForDNAFragment (String fragment)`: $O(n^2)$
- `GenHocDNADatabase.removeDuplicateDNASequences()`: $O(n^4)$

6.5 O-Notation for the project

`testLookingForCrimeSceneCodes(int howMany)` – $O(n^4)$

`removeDuplicateDNASequences()` – $O(n^4)$

The estimate O-notation is $O(n^4)$ because there are two methods that the O-notation in the worst case is $O(n^4)$ (`testLookingForCrimeSceneCodes(int howMany)` & `removeDuplicateDNASequences()`)

6.6 O-Notation for

- **Simulation. removeDuplicateDNASequences ()** - $O(n^4)$
- **Simulation. containsWholeDNASequence ()** - $O(n^2)$
- **Simulation. searchForDNAFragment ()** - $O(n^2)$

7. Warp Factors

Average:

DNALOOKUPS WARP = 684

Crime scenes WARP = 154

fragment Search WARP = 1

Duplicates WARP = 4

8. Changes and why

Original Code	After changes code	Reasons for changes
<pre>public void addCrimeScene(String whereID) { if(this.crimeScenes.contains(whereID)) return ; // must be this.crimeScenes.add(whereID); }</pre>	<pre>public void addCrimeScene(String whereID) { if (mapScenes.get(whereID) != null) { return; } mapScenes.put(whereID, whereID); crimeScenes.add(whereID); }</pre>	Using an hashMap as well as the ArrayList. The hashMap is mainly to check if there is already the String whereID in the data structure. With the hashMap this takes constant time $O(1)$, and in the ArrayList the method contains takes linear time $O(n)$. The ArrayList is needed to the get function, on the method getRandomCrimeScene (Random dice), continues to take constant time.
<pre>public boolean linkedToCrimeScene(String whereID) { assert crimeScenes !=null ; for(String it : crimeScenes) { if(whereID.equals(it))return true ; } return false ; }</pre>	<pre>public boolean linkedToCrimeScene(String whereID) { assert mapScenes != null; boolean result = false; if (mapScenes.get(whereID) != null) { result = true; } return result; }</pre>	The method of checking if the data structure as already the dna sequence with the String whereID, in the hashMap takes constant time $O(1)$ and when using a LinkedList,

<pre> }</pre>		<p>the O-notation is $O(n^2)$, because it needs to iterate through every element of the list and check if its dna sequence is equal to the one passed as parameter.</p>
<pre> public boolean searchForDNA(String dna) { for(DNASequences it: database) { if(it.sameDNASequences(dna))return true ; } return false ; } </pre>	<pre> public boolean searchForDNA(String dna) { boolean result = false; if (db.get(dna) != null) { result = true; } return result; } </pre>	<p>The method of checking if the database as the DNASequences with the string dna passed as parameter had an O-notation of $O(n^2)$, but when using an hashmap this method's O-notation changes to $O(1)$.</p>
<pre> public int removeDuplicateDNASequences() { List<DNASequences> duplicates = new ArrayList<DNASequences>() ; for(int a = 0 ; a < database.size();a++) { DNASequences thisOne = database.get(a); for(int b = a+1 ; b < database.size(); b++) { DNASequences otherOne = database.get(b); assert thisOne != otherOne ; // skip self if(thisOne.sameDNASequences(ot herOne.getDNASequences())) { duplicates.add(otherOne); } thisOne.addCrimeSceneFrom(ot herOne); } } for(DNASequences dup : duplicates) { database.remove(dup); } return duplicates.size(); } </pre>	<pre> public int removeDuplicateDNASequences() { List<DNASequences> duplicates = new LinkedList<DNASequences>(); Set<DNASequences> setDups = new HashSet<>(); for(int i = 0; i<database.size(); i++) { DNASequences dna = database.get(i); if(!setDups.add(dna)) { duplicates.add(dna); } } for (DNASequences dup : duplicates) { database.remove(dup); } return duplicates.size(); } </pre>	<p>Using an HashSet to deal with the duplicates changes the O-notation of the method from $O(n^4)$ to $O(n^2)$</p>
<pre> public void testRemoveDuplicateDNASequences(final int DUPLICATES) </pre>	<pre> public void testRemoveDuplicateDNASequences(final int DUPLICATES) { </pre>	<p>Changes just to make the code cleaner and to</p>

<pre> { int before = database.size(); assert DUPLICATES < database.size(); List<DNASequence> duplicates = new ArrayList<>() ; for(int a = 0 ; a < DUPLICATES ; a++) { DNASequence dup; do { DNASequence existing = pickRandomSequence(dice); dup = makeDNASequence(existing.getDnaSequence()); } while(duplicates.contains(dup)== true); duplicates.add(dup); } for(DNASequence dup: duplicates) { addDNASequence(dup); } assert database.size() == before + DUPLICATES : "Error making duplicates" ; int removed = removeDuplicateDNASequences(); assert removed == DUPLICATES : "Number of duplicates found wrong Removed=" + removed + " before " + before + " " + DUPLICATES + " " + database.size() ; assert database.size()== before ; } </pre>	<pre> int before = database.size(); assert DUPLICATES < database.size(); Map<String, DNASequence> duplicates = new HashMap<>(); for (int a = 0; a < DUPLICATES; a++) { DNASequence dup; do { DNASequence existing = pickRandomSequence(dice); dup = makeDNASequence(existing.getDn aSequence()); } while (duplicates.containsKey(dup.getDn aSequence())); duplicates.put(dup.getDnaSequenc e(), dup); addDNASequence(dup); } assert database.size() == before + DUPLICATES : "Error making duplicates"; int removed = removeDuplicateDNASequences(); assert removed == DUPLICATES : "Number of duplicates found wrong Removed=" + removed + " before " + before + " " + DUPLICATES + " " + database.size(); assert database.size() == before; } </pre>	<p>delete the use of unnecessary for loops. The changes made do not change the time complexity of the method.</p>
<pre> public boolean testLookingForFragementPositive (int kFragementPositiveTests) { Random rn = this.dice ; boolean result = true ; for(int a = 0 ; a < kFragementPositiveTests ; a++) { DNASequence existing = database.get(rn.nextInt(database.size())); String existString = existing.getDnaSequence(); </pre>	<pre> public boolean testLookingForFragementPositive(i nt kFragementPositiveTests) { Random rn = this.dice; boolean result = true; for (int a = 0; a < kFragementPositiveTests; a++) { DNASequence existing = database.get(rn.nextInt(database.s ize())); String existString = existing.getDnaSequence(); existString = existString.substring(rn.nextInt(exi stString.length() - 5), </pre>	<p>The use of the List foundList, given by the searchForDNAFragement method, was unnecessary because it was only used to check if its size was equal to zero. This was changed to a simple Boolean and a loop through the database that returns true when the DNASequence</p>

<pre> existString = existString.substring(rn.nextInt(existString.length()-5), existString.length()); List<DNASequence> foundlist = searchForDNAFragment(existString) ; if(foundlist.size() == 0) { System.out.println("You made a mistake - could not find DNA *FRAGMENT* which existed "); result= false ; } return result ; } } </pre>	<pre> existString.length()); boolean found = false; for (DNASequence it : database) { if(it.getDnaSequence().contains(exi stString)) { return true; } if (!found) { System.out.println("You made a mistake - could not find DNA *FRAGMENT* which existed "); result = false; } return result; } } } </pre>	<p>contains the DNA fragment. The changes made do not the change the time complexity of the method.</p>
<pre> public void readyForSelfTest() { prepareDataBase(); } </pre>	<pre> public void readyForSelfTest() { loadSimulatedDNAFromDatabase() ; } </pre>	<p>The original method was calling several different methods that only call other methods. Although it takes constant time $O(1)$, this is bad coding and bad practice because it has no purpose and can lead to unnecessary mistakes. The changes made do not the change the time complexity of the method.</p>
<pre> public void resetDatabase() { this.database = new ArrayList<DNASequence>() ; } </pre>	<pre> public void resetDatabase() { this.database = new ArrayList<DNASequence>() ; this.db = new HashMap<>(); } </pre>	<p>This method was only changed to reset the database ArrayList as well as the database HashMap. The changes made do not the change the time complexity of the method.</p>
<pre> public int howManySequencesForCrimeSce ne(String crimeSceneCode) { assert crimeSceneCode != null ; int howMany = 0; for(DNASequence seq: database) { </pre>	<pre> public int howManySequencesForCrimeScen e(String crimeSceneCode) { assert crimeSceneCode != null; int howMany = 0; for (DNASequence seq : database) { if (seq.linkedToCrimeScene(crimeSce neCode)) { howMany++; </pre>	<p>This method is only called by the method testLookingForCrimeSceneCodes (int howMany) and it is called to check if its result is greater than zero. Because of that the code was</p>

<pre> if(seq.linkedToCrimeScene(crimeSc eneCode)==true)howMany+= 1; } return howMany ; } </pre>	<pre> } if (howMany >= 1) { return howMany; } } return howMany; } </pre>	<p>changed to return the result as soon as its value is greater than one. This way it does not need to loop through the entire database. The changes made do not the change the time complexity of the method.</p>
<pre> public boolean lookForPositiveDNASeq(int howMany) { assert howMany > 0 ; boolean result = true ; // do not change 50000 Dr Andrew Chemist Chief technology officer GenetHoc Ltd. for(int a = 0 ; a < howMany ; a++) { DNASequence existing = pickRandomSequence(dice); String existString = existing.getDnaSequence(); if(searchForDNA(existString) == false) { System.out.println("You made a mistake - could not find DNA which existed "); result= false ; } } return result ; } </pre>	<pre> public boolean lookForPositiveDNASeq(int howMany) { assert howMany > 0; boolean result = true; // do not change 50000 Dr Andrew Chemist Chief technology officer GenetHoc Ltd. for (int a = 0; a < howMany; a++) { DNASequence existing = pickRandomSequence(dice); String existString = existing.getDnaSequence(); if (searchForDNA(existString) == false) { System.out.println("You made a mistake - could not find DNA which existed "); result = false; return result; } } return result; } </pre>	<p>This method only returns a Boolean. With the original code the method was iterating through the entire database. Adding a return as soon as the if statement is valid, would stop the method to loop through every element of the database. The changes made do not the change the time complexity of the method.</p>
<pre> public boolean lookForNegativeDNASeq(int howMany) { assert howMany > 0 ; boolean result = true ; // do not change 50000 Dr Andrew Chemist Chief technology officer GenetHoc Ltd. for(int a = 0 ; a < howMany ; a++) { if(searchForDNA("Dr Andrew Chemist Chief technology officer GenetHoc") == true) { System.out.println("You made a mistake - could not find DNA which existed "); result= false ; } } return result ; } </pre>	<pre> public boolean lookForNegativeDNASeq(int howMany) { assert howMany > 0; boolean result = true; // do not change 50000 Dr Andrew Chemist Chief technology officer GenetHoc Ltd. for (int a = 0; a < howMany; a++) { if (searchForDNA("Dr Andrew Chemist Chief technology officer GenetHoc") == true) { System.out.println("You made a mistake - could not find DNA which existed "); result = false; } } } </pre>	<p>This method only returns a Boolean. With the original code the method was iterating through the entire database. Adding a return as soon as the if statement is valid, would stop the method to loop through every element of the database.</p> <p>The changes made do not the change the time complexity of the method.</p>

}	return result;	
	}	
	}	
	return result;	
	}	

9. Screen Dump of Profiler after the New Code

main		4,648 ms (100%)
genethoc.StudentSimulation.main (String[])		4,648 ms (100%)
genethoc.GenHocDNADatabase.everythingWhichShouldBelImproved ()		4,647 ms (100%)
genethoc.GenHocDNADatabase.selfTest ()		3,623 ms (78%)
genethoc.StudentSimulation.readyForSelfTest ()		2,538 ms (54,6%)
genethoc.GenHocDNADatabase.testDatabaseIntegrity ()		1,085 ms (23,3%)
Self time		0,082 ms (0%)
genethoc.GenHocDNADatabase.testLookingForCrimeSceneCodes (int)		633 ms (13,6%)
genethoc.StudentSimulation.howManySequencesForCrimeScene (String)		630 ms (13,6%)
genethoc.StudentSimulation\$StudentDNASequence.linkedToCrimeScene (String)		372 ms (8%)
Self time		257 ms (5,5%)
genethoc.GenHocDNADatabase.getSomeRandomCrimeScenes (int)		2,41 ms (0,1%)
Self time		1,4 ms (0%)
genethoc.StudentSimulation.testRemoveDuplicateDNASequences (int)		364 ms (7,9%)
genethoc.GenHocDNADatabase.makeDNASequence (String)		331 ms (7,1%)
genethoc.GenHocDNADatabase.makeRandomDNASequence ()		227 ms (4,9%)
genethoc.DNASequence.addRandomCrimeLocations (genethoc.DNASequence, java.util.Random)		103 ms (2,2%)
Self time		0,493 ms (0%)
genethoc.StudentSimulation.removeDuplicateDNASequences ()		31,4 ms (0,7%)
Self time		1,32 ms (0%)
genethoc.GenHocDNADatabase.pickRandomSequence (java.util.Random)		0,466 ms (0%)
genethoc.StudentSimulation.addDNASequence (genethoc.DNASequence)		0,434 ms (0%)
genethoc.GenHocDNADatabase.testLookingForDNASequence (int)		13,4 ms (0,3%)
genethoc.GenHocDNADatabase.prepLookingForDNASequence (int)		13,4 ms (0,3%)
genethoc.StudentSimulation.lookForPositiveDNASeq (int)		9,68 ms (0,2%)
genethoc.StudentSimulation.lookForNegativeDNASeq (int)		3,67 ms (0,1%)
Self time		0,105 ms (0%)
Self time		0,021 ms (0%)
genethoc.GenHocDNADatabase.testLookingForFragement (int)		11,2 ms (0,2%)
genethoc.StudentSimulation.testLookingForFragementPositive (int)		11,2 ms (0,2%)
Self time		0,020 ms (0%)
Self time		0,648 ms (0%)
Self time		0,283 ms (0%)
genethoc.StudentSimulation.<clinit> ()		0,022 ms (0%)