**1.INTRODUCTION**

A protein often interacts with another protein to perform its function. Hence, to understand the molecular basis of biological processes, it is important to study which proteins interact. Public databases such as **BioGrid** ( Biological General Repository for Interaction Datasets )collect protein-protein interaction (PPI) data. Protein-Protein interaction is the physical contacts established between two or more proteins as a result of biochemical events.

**BioGRID** has data compiled through comprehensive curation efforts and has a total number of 749,199 non-redundant PPIs. These interactions involve 63,007unique genes and represent 30 model organisms.

* 1. **PROBLEM STATEMENT**

1. The first problem of the BioGrid is definitely the size of the MITAB file which is about 599554 KB(585MB) and the amount of space taken when it is opened would consume all the main memory if opened. To access and manipulate a file it must be in memory entirely.
2. Secondly searching for a particular interaction could take up a large amount of time because of the large number of interactions.
3. There is no direct access to a particular interaction data. Need to linearly search through all the 749,199 interactions (749,199 lines) to find it in the worst case scenario. The Time Complexity of the linear search algorithm is O(n).
4. Compared to databases, the file cannot handle indexing task, therefore slowing the search since there are no structural relationships.
   1. **AIMS AND OBJECTIVES**

The aims and objectives are:

1. To perform data exploration to get a “feel for the data”. This includes researching about the structure of the file, how it is organized and the meaning of the fields.
2. To design a data structure to facilitate data programming. This requires us to use an adapted data structure to store the fields therefore reducing programming effort such as array or list.
3. To be able to sort the BioGrid file. This requires the use of sorting algorithms and minimizes the time complexity using the most appropriate one such as the Quick Sort algorithm.
4. To be able to search for interactions between genes and output them to a file. Searching may take some time if a linear search algorithm. Therefore the use of selective algorithm is crucial to obtain the list of interactions.
   1. **DISTRIBUTION OF TASK**

There are 2 team members Haidar and Hansynee in our group and each one will be given specific tasks. we would break the task equally so as each of us share the same work load.

Hansynee would be doing the data analysis, how the data is organized, what type of data, how many interactions and the fields necessary.

Haidar would be designing the algorithm and discuss, implement it together and analyse the time complexity.

We would carry out an incremental approach to solve our problem, starting at the very first week and carrying out the report alongside, adding a new complexity each week. Thus the very first week would be more basic and as we move, it would become more complex and would require a more complex data structure as well.

**2.0 ANALYSIS OF DATA SET**

In this section we will try to analyse the BioGRID.mitab file to understand the structure of the data set, the data arrangements, the meaning of each columns and delimiters used to separate each field.

**2.1 DESCRIPTION OF DATA SET**

It is very important to know the content of the BioGRID to be able to extract required information and do the relevant actions.

**Column Definitions(**Biogrid.wiki)

The MITAB format is part of the PSI-MI 2.5 standard. It is derived from the TAB 1.0 format provided by the BioGRID. MITAB only described binary interactions with one pair of interactions per **row**. Columns are separated by **tabulations**.

The complete descriptions of the fields is available at <http://wiki.thebiogrid.org>/doku.php/ psi\_mitab\_file

|  |
| --- |
| 1. Unique identifier for interactor A. |
| 1. Unique identifier for interactor B. |
| 1. Alternative identifier for interactor A- the official gene symbol. Representation as databaseName:identifier. Multiple identifiers separated by “|”. |
| 1. Alternative identifier for interactor B. Same stucture as Point 3. |
| 1. Aliases for A, separated by “|”. Representation as databaseName: identifier. Multiple identifiers separated by “|”. |
| 1. Aliases for B. Same stucture as point 5. |
| 1. Interaction detection methods. |
| 1. First author surname(s) of the publication(s). |
| 1. Identifier of the publication . |
| 1. NCBI Taxonomy identifier for interactor A |
| 1. NCBI Taxonomy identifier for interactor B. |
| 1. Interaction types, |
| 1. Source databases and identifiers. |
| 1. Interaction identifier(s) . |
| 1. Confidence score |

The first line of the file contains the headings for the fields and is preceded by a hash tag ‘#’ and is a such:

“#ID Interactor A ID Interactor B Alt IDs Interactor A Alt IDs Interactor B Aliases Interactor A Aliases Interactor B Interaction Detection Method Publication 1st Author Publication Identifiers Taxid Interactor A Taxid Interactor B Interaction Types Source Database Interaction Identifiers Confidence Values”

All the columns are mandatory, therefore columns with empty values are filled with “-“.

Some gene names are in uppercase and lower case and the combination of the 2.They are preceded by ‘entrez gene/locuslink:’ and there may be more than one gene available.

biogrid:72804|entrez gene/locuslink:Dip1|entrez gene/locuslink:Dmel\_CG15367 biogrid:72804|entrez gene/locuslink:Dip1|entrez gene/locuslink:Dmel\_CG15367

**2.2 DESCRIPTION OF THE SYSTEM**

Access to unified datasets of protein and genetic interactions is crucial for interrogation of gene/protein function and analysis of global network. Therefore a rapid search and retrieval of interaction data is a must. Without using an appropriate program to access the valuable data, it will take a very long time to scroll past the BioGRID file and to search for a specific interaction.

The BioGRID distribute a comprehensive collection of physical and genetic interactions. With increasing number of updates we must be able to devise ways in order view an interaction.

**2.3 FUNCTIONAL REQUIREMENTS**

The system should be able to:

1. To output a sorted list of distinct genes.
2. To output a list of interactions based on an input genes
3. To search an interaction based on the alias input.

**2.4 NON-FUNCTIONAL REQUIREMENTS**

The non-functional requirements are the quality requirements:

Performance/response time-The system should be as efficient as possible and should take into consideration any possible overhead.

Testability-The system shall be tested for errors and a variety of algorithm must be critically analysed to find the optimal one.

Usability- An appropriate interface shall be developed to allow the user to enter the gene of interest via keyboard and the choices to be made. A command prompt mainly to enter the choices. A menu of options can be used.

**3.DESIGN**

Sorting algorithm: To obtain a list of sorted distinct Gene Names in ascending order, the Entrez Gene/Locuslink will be used as the sorting parameter. A number of algorithms can be used to sort the list of Gene Names, but the optimal one will be determined by their time complexity.

**3.1 ALGORITHM**

The first part requires us to output a file of distinct gene names in ascending order and without duplicates. Hence these are the algorithm that will be used.

1. CODE FOR INSERTING GENE NAME IN GENEARRAY

|  |  |
| --- | --- |
| while not eof | Loop through file to indentify each gene Name |
| GeneA=ObtainGenes(); |  |
| If(not duplicate(GeneA) | Check if gene already enterred |
| InsertInArray((GeneA) | if not entered , insert in array |
| end if |  |
| End while |  |

1. INSERTION SORT ALGORITHM

|  |  |
| --- | --- |
| for i=1 to N-1 | Loop through array to place every gene name in its position |
| Key=GeneArray[i] |  |
| J=i-1 |  |
| While(j>=0) AND (GeneArray[j]>key) |  |
| GeneArray[j+1]=GeneArray[j] | Shifting array content to the right to insert gene name in correct pposition |
| J=j-1 |  |
| End while |  |
| GeneArray[j+1]=key |  |
| Endfor |  |

1. BINARY INSERTION SORT ALGORITHM

|  |  |
| --- | --- |
| Int size=0 | Current size reached |
| For i=0 to N-1 | Loop through array to insert gene name in correct position |
| Key=GeneArray[i] |  |
| Pos=BinarySearch(key) | the position to insert gene name,found using binary search |
| If GeneArray[pos]==Key | Check for duplicates |
| Exit |  |
| For(i=size to pos) | Shifting array content to right starting from the last item |
| GeneArray[i+1]=GeneArray[i] |  |
| GeneArray[pos]=Key |  |
| Size=size+1 | Gene inserted,,increment size |
| End for |  |
|  |  |

|  |  |
| --- | --- |
| BinarySearch(low,high,key) |  |
| If (low==high) |  |
| Return low |  |
| else |  |
| Mid=low+(low+high)/2 | List broken into 2 |
| If Key>GeneArray[mid] |  |
| Return BinarySearch(mid+1,high,key) | Right half of array because |
| Else if key<GeneArray[mid] |  |
| Return BinarySearch(low,mid,key) | Left half of the array |
|  |  |
| Return mid |  |
|  |  |

1. Quick Sort

|  |  |
| --- | --- |
| Int partition(L,R){ |  |
| Int I=L; |  |
| Int J=R; |  |
| While(GeneArray[I]<GeneArray[R]) | Ensure that GeneNames before GeneArray[R] are smaller |
| I=I+1 |  |
| While (GeneArray[J]>=GeneArray[R]){ | Ensure that GeneNames After GeneArray[R] are Greater |
| J=J-1 |  |
| If(I<J) |  |
| SWAP(I,J) | Swap items to make List in correct Order |
| else |  |
| SWAP(I,R) } | Separate List into two |
| Return I |  |
|  |  |
| QuickSort(GeneArray[],int L,int R){ |  |
| If(R>L) |  |
| X=partition(GeneArray,L,R) |  |
| QuickSort(GeneArray,L,X-1) |  |
| QuickSort(GeneArray,X+1,R) |  |
| END IF |  |
| } |  |
|  |  |

The second part requires us to output a file of interactions. For that part we will use hashing to retrieve a particular interaction. Thus the time complexity of hashing is O(1) in the best case, thus the quickest search compared to binary search and linear. Below shows the algorithm:

|  |  |
| --- | --- |
| LinkedList[] GeneArray=new LinkedList[size]; | The logic behind is to use an array of linked list with head as the |
| HashInsert(GeneA,GeneB){ | Alt\_ID for interactor A and the rest as the Other genes interacting |
| pos=hash(GeneA); | With it. |
| If (GeneArray[pos] is null) |  |
| GeneArray[pos].insertfirst(GeneA); | Hence if the position is empty,therefore no genes not yet entered. |
| GeneArray[pos].insertLast(GeneB); | Insert as head GeneA and the other as the interacots |
| } |  |
| Else{ |  |
| GeneArray[pos].insertLast[GeneB] | If an entry is already made,then no use to enter GeneA since it |
| } | Already exist.We only have to insert geneB. |
| } |  |
|  |  |
|  |  |

The algorithm above show the insertion part. The same principle would apply to the search.

|  |  |
| --- | --- |
|  |  |
| Void Search(geneA){ |  |
| Pos=hash(GeneA); | We have to hash the gene name,look at the position |
| If (found) | If found then display interaction. |
| Display interactions |  |
| Else |  |
| Print gene interactions not found | Print genes not found |
|  |  |
| } |  |
|  |  |

**3.2 DESIGN ISSUES**

The algorithms listed in section 3.1 is in its most basic form and therefore must be modified to suit its use. In the following sections we will analyse each algorithms performance and explain how we came to these. The algorithms will be analysed to find their time complexity.

**3.2.1 PERFORMANCE**

3 elementary algorithms were studied to sort the gene names namely SELECTION SORT, INSERTION SORT and QUICKSORT.

1. The selection sort has a time complexity of O(N^2) .The number of iterations is N in the first call the N-1, N-2 and so on. This evaluates to the arithmetic series (1+2+3+…+ (n-1)+n) therefore leading to N(N+1)/2 , giving us O(N^2).Since we are concerned with performance , this algorithm is therefore not suitable for large data.
2. Next, we came up with insertion sort which is also an elementary algorithm and a brute force as well. it is slightly better than selection sort in the sense that its best case is O(n) when the item to be inserted is to be the last in the array. However the worst case is O(N^2) for number of comparisons increases with each new item. Same explanation as the above. We might as well incorporate the ‘search for duplicate’ in the algorithm and not pre fill the array since the insertion sort takes one item at a time and put it in its place.
3. The binary insertion sort is an improvement of the insertion sort. Since the list is already being sorted, instead of searching from the beginning of the file to the end, we only have to apply a binary search to the existing array to find the position to insert our Gene. Asymptotically the binary insertion sort in no better than the Insertion sort but it does fewer comparisons. Since we have to search for duplicates first, this makes our task less time consuming.
4. The Quick Sort algorithm Is also a brute-force algorithm and uses a recursive approach to sort the GeneArray . The worst-case is O(N^2) and its average case is O(nlog(n)). However the search for duplicate algorithm has time complexity of O(N^2).Therefore methods must be devised to reduce the time complexity hence in our case by the use of Hashing to HashInsert the genes. In case of duplicates they would hash at the same index. The Time complexity in the best case for hashing is O(1).
5. Considering the first and the second part of the question, it is thought to be wise to combine both the methods from the sorting and the searching to make efficient use of the algorithm. Hence in our case use hashing to fill, check duplicates in the array then compacting it for the sort and use of hashing for search.

**3.2.2 ROBUSTNESS**

The system can deal with irregularities in the case when there are no aliases. The program must recognize it and should not enter it in the systems. Blank entries have ‘-‘. Concerning inputs, the only ones will be the gene name so as to output the interactions.

However if there has been a mistake in the file itself (gene name not correctly written) then there will be a mistake and we cannot correct it.

**3.3.3 ERROR AND EXCEPTION HANDLING**

Care must be taken in case of errors. If the path is not well specified, then an exception has to occur. Since the input is the gene name, if a wrong name is entered, appropriate messages must be displayed to the user.

Try catch block have to be used in this case, in case file is not found.

**4. IMPLEMENTATION AND EVALUATION**

In this topic we will discuss mainly how the algorithms are implemented to meet the objectives and to attain the functional and non-functional requirements.

**4.1.TOOLS AND ENVIRONMENT USED**

Java is an elegant language combined with a powerful and well designed set of API hence java will be used as platform. “Write once, run anywhere”.

The Java program reads lines, depending on the newline, while the C++ program reads white space delimited "words", which is a little extra work.

Eclipse is used as the IDE.

Low level languages like C are fast but has certain drawbacks, for example while reading file we would need to know the position of “\n” to mark the end of line. Java does it automatically.

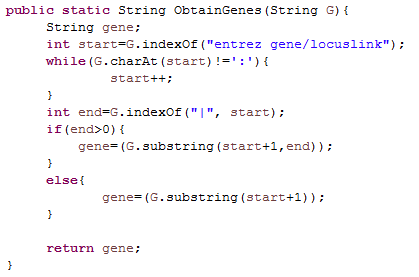
**4.2 IMPLEMENTATION CHALLENGES**

The challenges that have been to implement the logic behind the code to search and input the genes in the array because initially our program worked on a linear check to detect any duplicate thus this would take a very long time as for each gene input it would have to check the entire array.

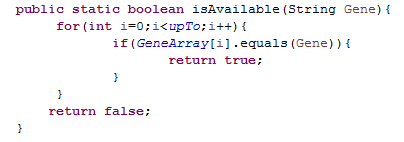
At first, we entered the whole details of the interaction but soon knew that we needed a certain level of abstraction since the file is very large and would run out of space if we loaded it in memory and it would lag. Thus we would only insert the official gene names and the aliases.

* 1. **SAMPLE CODE OF MAIN PIECES OF LOGIC**

|  |
| --- |
| this code serves to extract the gene name from the entrez gene /locuslink.  It locates the entrez gene/locuslink and return the gene name. |

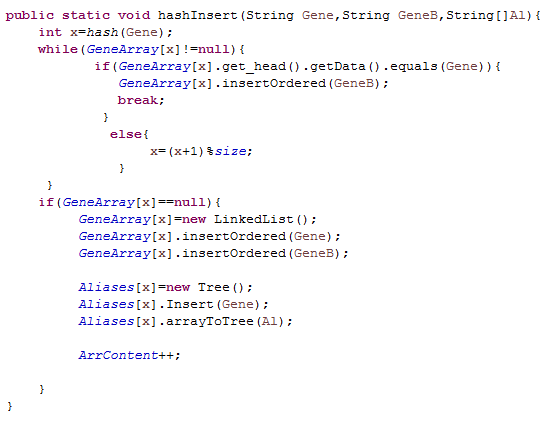
****

|  |
| --- |
| This is the code for checking if gene name already exists. However the time complexity is O(N^2).Therefore other methods should be used. This code **cannot** be used as a result |

****

|  |
| --- |
| This function call insert gene name into the hash table..The first time ,it is call with gene A , the geneB and the second time it is reversed. This is because each gene will have its own list of interactors. A->B,A->C, A->D,B->X.  In A’s list there will be B,C,D whereas in B there will be A,X.(‘->’ means interact) |

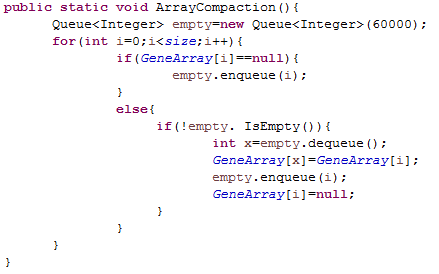
|  |
| --- |
| This piece of code is an improvement of the above one and consist of ‘Hash Inserting’ the gene. Hence when a duplicate name is hashed, it will be hashed at the **same** location thus will be marked as an **interaction**. Thus the other gene (InteractorB) is inserted in the linked list,with the head representing interactorA.  The Other nodes in the list will be the genes interacting with interactorA.  However when a distinct gene is hashed at the **same** index, **linear probing** Is used to find a free position.  Hash () is a function which hashes the gene name and return a ‘unique’ index. |

****

|  |
| --- |
| In the above code, a tree is used to store the aliases, hence while inserting the list of interaction in the hash table, the tree of aliases is stores in the same index in another array.(not to confuse, there is one array of linked list for the interactions and one array of trees for the aliases). |

|  |
| --- |
| Of course since **hashing** is used, **all** the functions that is associated to it is used like finding the next prime number to use as size of array to minimize collisions has been used. Furthermore the insertOrdered() method inserts the gene names in the linked list in alphabetical order. |

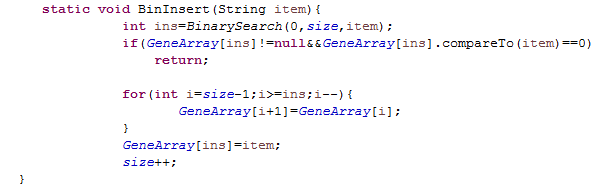
|  |
| --- |
| -The ArrayCompaction() function seeks to eliminate holes left when hashing so as to be able to apply the sorting algorithm without which it would not be possible.  -Its functionality is based on the rationale that the first blank encountered must be filled first, hence a FIFO Queue.  -Furthermore it does not create any duplicate array to solve the problem.  The complexity of this algorithm is O(N). |

****

*quicksort*(GeneArray,0,*ArrContent*-1)

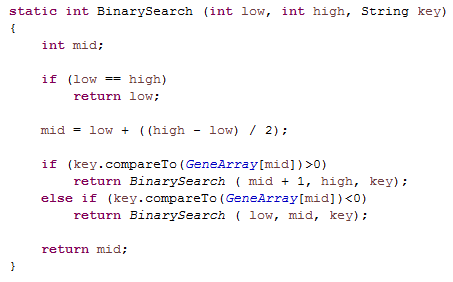
**An alternative way to sort the names**

|  |
| --- |
| The binary insertion is an improvement of the insertion sort in the sense that it makes less comparison to identify which item is a duplicate or where to insert item.  It then shift content of array just after the change to the right |

****

|  |
| --- |
| These are the implementations of the different sorting algorithm, only one will be retained for the final application. Implementations have been carried out to check the time taken, complexity of the different codes. |

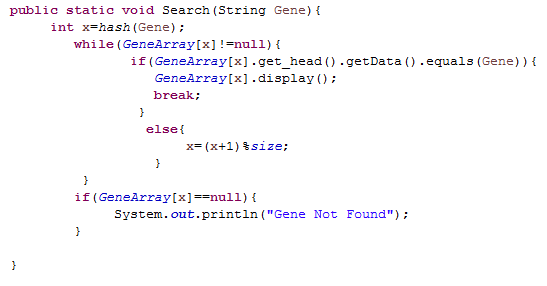
|  |
| --- |
| The binary search keeps on dividing the list into 2 thus narrowing the search area.  Thus it takes O(log n) to search for a particular item or index to enter |

****

|  |
| --- |
| After having entered the genes in the array, it is then sorted using the **QuickSort** Algorithm. This is given is the design section code listing 4.Since it is similar to the implemented one it is not shown. |

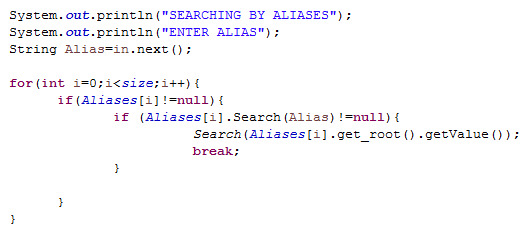
**THE SEARCHING PART**

|  |
| --- |
| The Search() consist of searching a interactions of a particular gene.  Here we hash the gene name the apply a variation of the linear probing to find the required interaction.  If we don’t find the gene, output gene not found. |

****

|  |
| --- |
| This Time we have to search through all the aliases to find the official gene name, and then search for it to find interactions.  Aliases[] is array of trees with each leaf representing aliases and the root representing the official gene name. Instead of using a list to store the aliases, a tree is used, hence quicker to search an alias. |

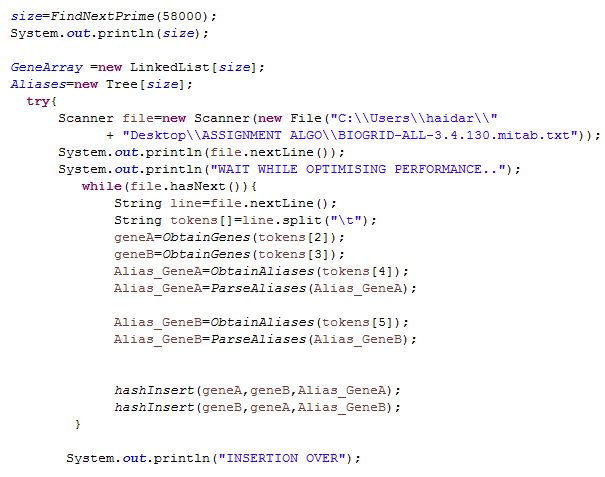
**SEARCHING BY ALIASES**

****

|  |
| --- |
| We need to search through all the trees in order to find the alias, therefore a linear search is used. but this is a better tradeoff since we only have to search through 56,000 array items to get the official gene name and use it to find the interactions(very fast since hashing is used). |

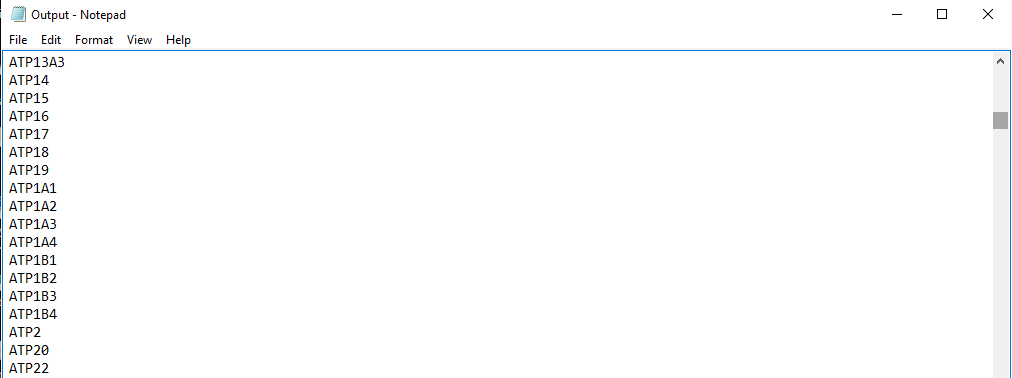
|  |
| --- |
| This is the main program where we call the different functions and read from the file.  We create an array of linked list for the interactions and array of trees for the alias.  We read every line of the file and hash insert them, |

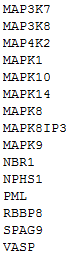
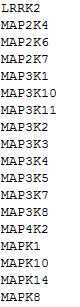
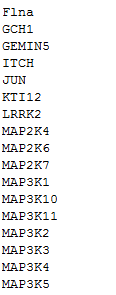
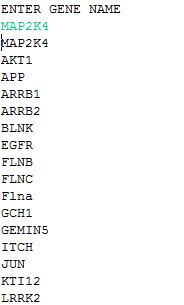
**The Main Program**

****

* 1. **SAMPLE SCREENSHOTS OF RESULTS**

1. Below shows screenshots of results for outputting a file of **distinct** gene names in ascending order.

****

1. Below shows screenshot of the search for interactions. The list can be verified at http://thebiogrid.org/112315/summary/homo-sapiens/map2k4.html****

**4.5.PERFORMANCE OF ALGORITHM**

|  |  |  |
| --- | --- | --- |
| **PART** | **Time Complexity** | **Running Time** |
| LOADING IN ARRAY AND CHECKING FOR DUPLICATES | Best case is O(1) for one hash, but for n inputs it is O(n) | An average has been performed and thus It takes 171,297 ms to load the contents into the array. It takes some time to feed into the array but once it is done, the searching is minimal. |
| SORTING THE ARRAY OF GENES | Since Quicksort has been used, as discuss earlier, the time complexity is O(nlog(n)) in the worse case. | It takes on average 1056 ms to sort the array. The elementary algorithm, if used would produce more delays. |
| SEARCHING FOR GENE INTERACTION | Since hashing is used, the best case is O(1).we just have to hash the gene name and retrieve. | The average time for this operation is 3795 ms. If an implementation of a tree structure is made, it could be greater since it has a time complexity of log (n). |

The overall running time may just be the time in loading the array. But this will be done only once (at the start) and hence the overall running time would be 3795 ms.

**5. DISCUSSION AND CONCLUSION**

5.1.ACHIEVEMENTS

The whole project has been implemented bearing in mind ‘is there a better way to do it’. This is how we arrived at the hashing which we would say is the main mechanism behind the system. In the beginning the issues was the checking of duplicates and insertion of the items. We may have thought of using tree structure but there was a better way to do it, improving the response time.

It may have been simpler to use the tree structure then traverse it in-order to output a sorted list, but considering the fact that we have to use the different sorting algorithm, we did forgo it.

5.2. CHALLENGES

Further avenues would include the **improvement** of the hashing function, to use **separate** **chaining** or better, **double** **hashing**. This would have into effect reducing the time taken to input the gene names. While the original idea debated was the use of a tree structure to hold the gene names and each node would be a linked list of interactions. Hence the hashing is an improved version, with quicker and more robust programming.

The veritable challenge was to be able to figure what duplication means. This would simply mean another interaction, hence populating a linked list with the other interactors, and hence each gene would have a list of interactors already populated while filling the array. Hence killing 2 birds with one stone: preparing the array for the second part(Searching) and filling the array for sorting.

**ISSUES**

One minor issue maybe the static nature of the hash table. This problem has been thought over and the use of a rehash function has been proposed. Thus to call the resize function we have to compare the size of the array and the number of inputs used.

|  |  |
| --- | --- |
| Resize(){ | This function would be called only when the limit is reached. |
| OldArray=GeneArray; |  |
| Size=FindNexPrime(2\*GeneArray.length); | We double the size of the array which would be more than enough |
| GeneArray=new LinkedList[Size] |  |
| ArrContent=0; |  |
| For(int i=0;i<OldArray.length;i++){ |  |
| If(OldArray[i]!=null) | Copy content of old array to new |
| GeneArray[i]=OldArray[i]; |  |
| } |  |
|  |  |
|  |  |
|  |  |
|  |  |

This function won’t be used again and again, it will be used only once.

It will also not affect the running time of the whole program since it is O(n).the program will function as it is.

This problem could also have been solved by entering the names in an arraylist first just to get the size then create an array of that size.

A tree structure would not encounter such problem, but all implementations have some advantages and some disadvantages and ours have been weighed against all.