GENE – satisfy research needs, an elegant way…

COMP3600 Final Project: Milestone-3

Jasmeen Kaur (UID - 6871990)
Date – 13 November 2020

Overview: -

Gene is an enterprise application software which is designed to meet the comprehensive needs of a medium-ranked *biotechnology* company in Australia – ***Ribo Biotechs Pvt. Ltd.*** Since *Gene* is an enterprise software, it aims to meet the needs of the company in a comprehensive manner. In its initial release it aims to implement 3 core functionalities required for both an efficient research practices and well organized administrivia at *Ribo Biotechs*.

Ms. Jen is a senior scientist and the Administrative Director of *Ribo Biotechs*. She is actively involved in the research fields of ***Comparative Genomics and Cancer Immunotherapy***. In order to conduct coherent research in a well-managed administrative environment at the workplace, she demanded the following functionalities required in the first release of Gene software: -

1. The current software used to conduct **Comparative Genomics** is surprisingly slow. Making DNA strands' comparisons faster and more accurate is the **most important** outcome which needs to be met by *Gene*.
2. Ms. Jen needs to create a team of **top 5 scientists** currently working in the company in order to undertake a robust *Cancer therapy research* in the coming year. She should be able to compare and rank scientists based on different criterion (such as *total research papers published by the scientist, total number of citations, overall rank in the company, research interests etc.*).
3. Currently, the employees' details are kept and accessed in a ***disorganized*** manner, wasting a lot of time which could have otherwise spent on conducting research. She would like to have a **single and secure point of access** for employees' data which can be accessed easily and in less time (*preferably constant amount of time*).

Let's have a look how *Gene* addressed the above problems in an elegant way ☺.

Functionality#1 – Comparative genomics: -

Aim: - To design an algorithm which outputs similarity (in %) between two DNA genomes up to a length of $\mathbf{2}\ \boldsymbol{million}$ by finding *Longest Common Subsequence* between two DNA strings.

Assumptions: -

1. A strand of DNA is made of 4 types of bases – *Adenine(A), Cytosine(C), Guanine(G), Thymine(T)*. Hence, we can express DNA as a string over the set – $\{A, C, G, T\}$.
2. DNA genomes are stored in ***.fa (fasta)*** **files** which are special file extensions used to store scientific information. Hence, the assumption is that user will ONLY input ***.fa (fasta)*** files as provided in `f1-test/` folder.
3. Size of stack in RAM is 1Mb.

Algorithm – Longest Common Subsequence (Dynamic Programming): -

Input: - Two ***.fa(fasta)*** files separated by a *space* of two species as given in `f1-test/` folder.

Output: - % Similarity between two input genomes measured by manipulating the length of length of Longest Common Subsequence between two DNA strings.

Optimized Approach: -

There exist many different algorithmic approaches to measure % similarity between genomes of different organisms such as: -

- Jaccard Similarity
- Jaccard Containment
- Number of conversions needed to convert one strand to another.

The approach which according to me is most **Space and Time efficient** (according to the concepts taught in this course) for calculating similarity among different genomes is – *Optimized Dynamic Programming*. In a brute-force approach to solving comparison problem, all subsequences of X will need to be enumerated and each subsequence is checked with that of Y, keeping track of the longest common subsequence found. Since X has 2^m substrings, this will lead to $exponential$ comparison time which is completely impractical for long strings.

The LCS problem has an $optimal - substructure\ property$ as the corresponding subproblems correspond to pairs of "prefixes" of the two input sequences. The optimal substructure is given by the following recursive formula: -

$$c\,[i,j] = \begin{cases} 0 & if\ i\ =\ 0\ or\ j\ =\ 0, \\ c\,[i-1,j-1]\ +\ 1. & if\ I,j\ >\ 0\ and\ xi\ =\ yi, \\ \max(c[I,j-1],c[i-1,j]) & if\ i,j\ >\ 0\ and\ xi\ !=\ yj \end{cases}$$

Note – $c[i,j]$: The length of LCS of X_i and Y_j.

Theoretical Complexity: -

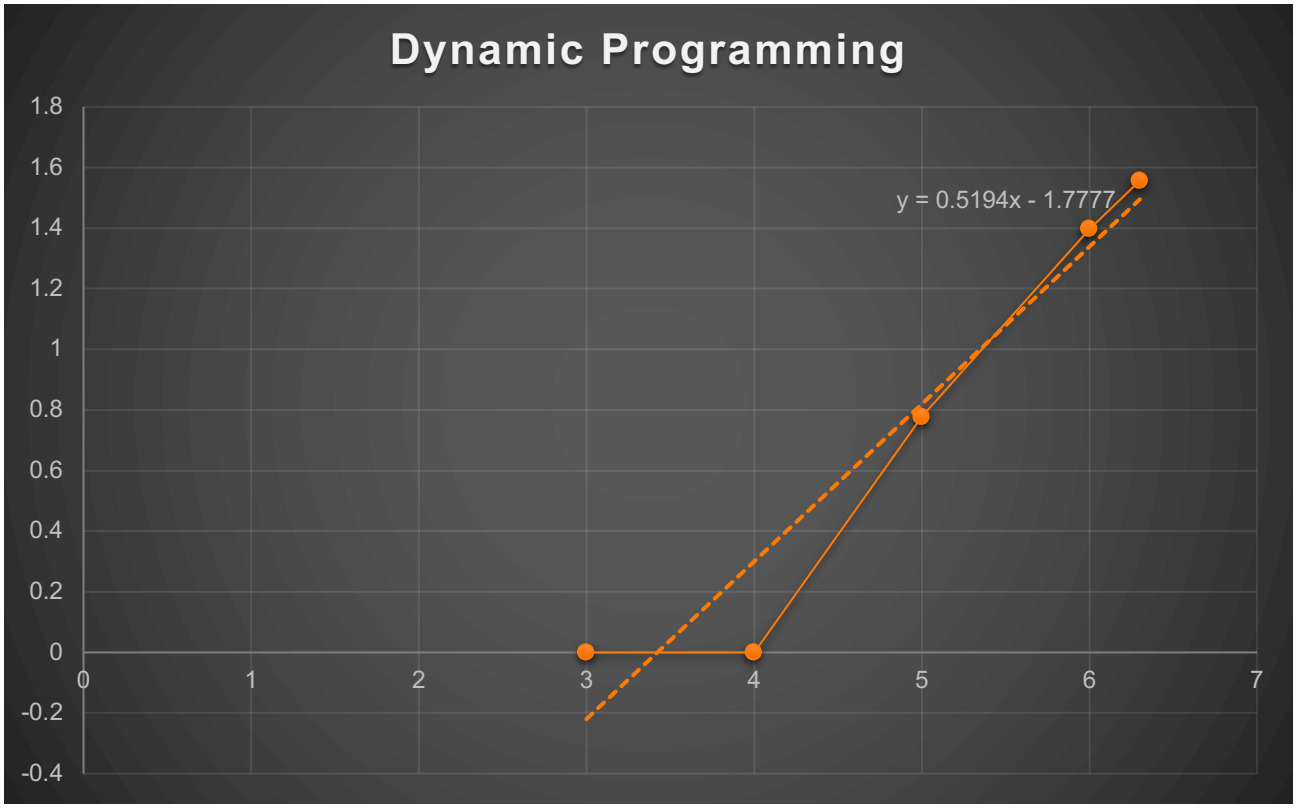
Time Complexity - The number of sub-problems constructing the DP-table is $\Theta(mn)$ and the time taken per subproblem is $\Theta(1)$. Hence the total time for the whole algorithm is $\Theta(mn)$. Since, number of comparisons take Quadratic time (as above), it will be time-consuming to compare strings of length as long as $2\ million$. I have made some optimizations to achieve results of DNA lengths equal to $2\ million$ in no more than 40 seconds. Since DNA genomes are NOT random, they possess a certain pattern of nucleotides which is constant for most of the species and rarely changes. Comparing same patterns of nucleotides across all species would be time-consuming, hence ignoring these constant sequences of nucleotides greatly helped me to reduce the total time from quadratic to linear degree.

Space Complexity – Since constructing a DP-table with dimensions $m * n$ for $m,n\ =\ 2\ million$ would require a huge table with 4,000,000,000,000 entries. In order to reduce this space, I have implemented a **smarter** Dynamic programming algorithm which takes **Linear space** and is NO less efficient. I have utilized the fact that each entry depends on only three other c table entries: $c\,[i-1,j-1]$, $c[I,j-1]$ and $c[i-1,j]$. Hence, I don't even need the whole 2-d array but just a single 1-D array to perform computations. Code was significantly less intuitive to write but it worth once implemented.

Empirical time complexity: -

Input_Size	Time_Taken(seconds)
1000	0
10000	0
100000	6
1000000	25
2000000	36

LOG(input_size)	LOG(time_taken)
3	0
4	0
5	0.77815125
6	1.397940009
6.301029996	1.556302501



Hence, the Experimental Time Complexity is close to $Linear$ which satisfies the Theoretical Complexity explained above.

Functionality#2 – Ranking Scientists based on different criterion:

Aim: - Rank scientists in the firm based on the criteria input by the user in setup time – $O(n)$ and access time ~ $O(1)$ (based on the assumption that employee details are NOT read and removed from the database frequently).

Assumptions: -

1. Size of stack in RAM is 1Mb.
2. Since implementation contains $structs$ of employees holding a total of 11 attributes for each employee, based on given memory constraints, maximum size of the input *.txt* file - $2000\ employees$.

3. User need NOT input any *.txt* file (from `f2-test/` folder) to perform testing on different faculty sizes. Size of *.txt* file is taken once the application STARTS and software itself handles the rest. **Different sizes could be chosen at the start to perform different tests**.

Algorithm – MaxHeap: -

Input: - **Size** of *.txt* file containing details of employees and scientists at the start of the application.

Output: - Employee details based on input criteria in descending order.

Optimized Approach: -

A naïve approach would be to first SORT based on input criteria like – *overall_ranking (or number of papers published, number of citations, research interests,qualifications etc.)* and then extract the top 5 scientists. This approach would take $O(n)$ time using Non-Comparison based sorting such as *Counting* and *Radix* sorts. But upon using Heap data structure, the complexity of finding the top 5 scientists is reduced to $O(log\ n)$ which is awesome!

Main operations implemented to extract n-top scientists (based on chosen criteria) in the most efficient way are as follows: -

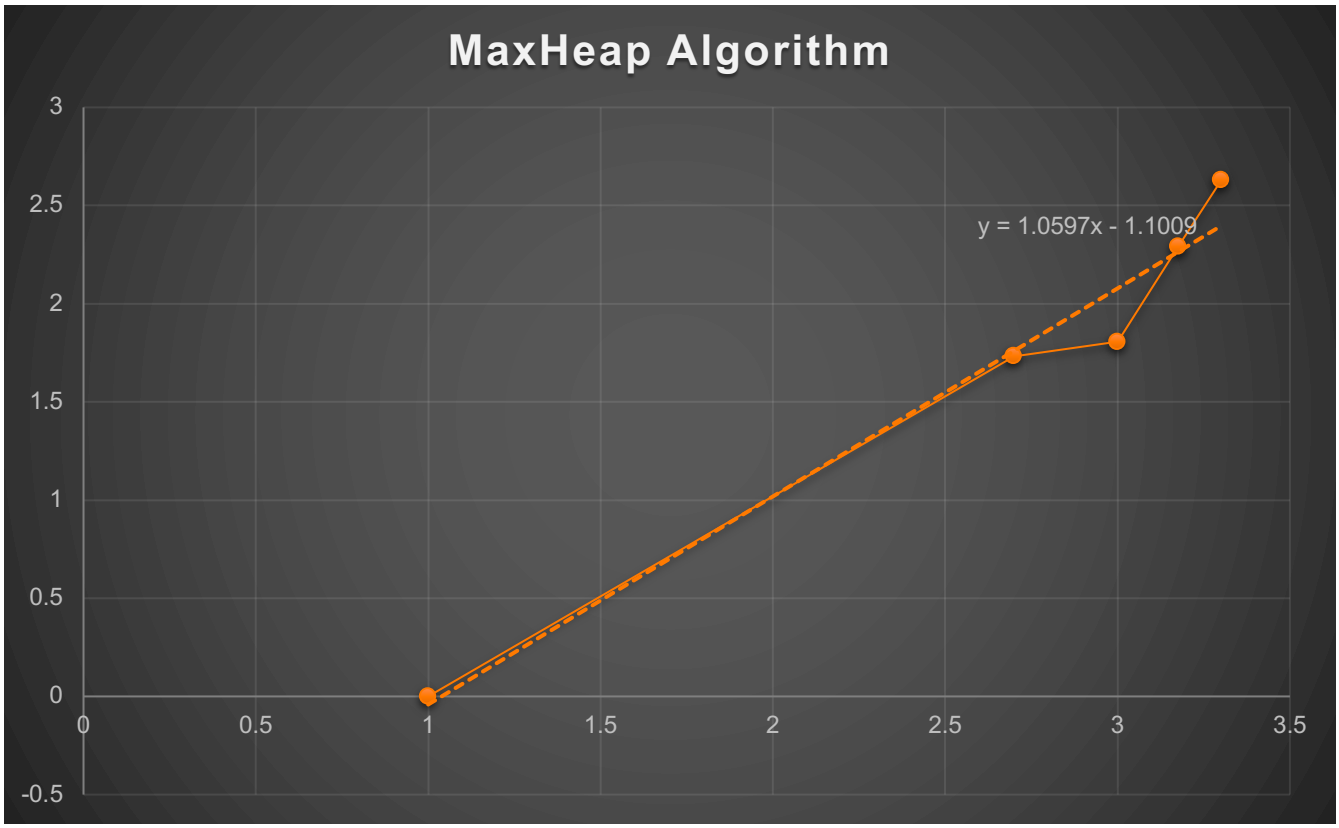
1. $Heapify$ to ensure properties of heap are being satisfied at all times during execution.
2. Inserting a node based on given criteria into the heap.
3. Extract the maximum or the minimum.

As algorithm only need to traverse the path of tree once, hence all of the above operations take $O(log\ n)$ time.

Empirical time complexity: -

Input_Size	Time_Taken(milli-secs)
10	0
500	54
1000	64
1500	195
2000	426

LOG(input_size)	LOG(time_taken)
1	0
2.698970004	1.73239376
3	1.806179974
3.176091259	2.290034611
3.301029996	2.629409599



Since the initial set up time of heap is $O(n)$ followed by $O(1)$ – look up time and $O(log\ n)$ – $heapification$, the slope of the trendline is ~ 1 which satisfies the theoretical analysis above.

Functionality#3 – Secure point of access (Universal Hashing with Open Addressing approach): -

Aim: - Provide an authentication portal for the user trying to access *Gene*.

Assumptions: -

1. Employees are NOT added frequently, hence size of the hash table is assumed to be static most of the time.
2. Size of stack in RAM is 1Mb.
3. Due to implementation of TWO Hash Table data structures in *Gene* (**one** for *authentication* using $User\ ID$ and $Password$ & **other** for searching an employee across whole database using $First\ name$ and $Last\ name$) and limited memory

constraints, employees' details file should no longer contain details of more than 2000 employees.

Input: - $User\_ID$ and $Password$ in the command line for authenticating access.

NOTE – $User\_ID$ and $Password$ should be taken from the correct `f2-test/facultyXXX.txt` file. **Example** – If user selected `2000` employees at the start of the application (for testing purposes), then ONLY $User\_ID$ and $Password$ present in `f2-test/faculty2000.txt` file will be valid to authenticate. By default, use – **`UserID:`** `kaurjas;` **`Password:`** `1234` in any case.

Output: - Authentication status in the terminal.

Optimized Approach: -

Main goal in the 3rd functionality is to have a **single and secure point of access** for employee details along with fast & efficient retrieval. Hash Tables are extremely efficient for this goal. Due to the above assumption of **infrequent** insertion and deletion of employee details, I have used `Open Addressing Hash Table` data structure which prevents from memory overhead of pointers in linked lists in `Chaining Hash Table data structures`. This also allows me to have much bigger hash table size to store a greater number of employees' structs. There are **TWO** hash tables in *Gene* for **authentication** and **searching** respectively as follows: -

Authentication – (OPEN ADDRESSING – 1):

`Key` `- UserID and Password;` **`Value/Satellite Data`** `- Employee Struct/record of attributes`

Employee Search - (OPEN ADDRESSING – 2):

`Key` `- First and Last name;` **`Value/Satellite Data`** `- Employee Struct/record of attributes`

Universal Hashing – I have chosen to implement Universal Hashing to make authentication and retrieval fast and secure. This approach is elegant due to the fact that there are NO malicious keys which can all hash to the same slot, yielding an average retrieval time of $\Theta(n)$. Universal Hashing yield provably good performance on average, no matter the input keys.

So, we begin by choosing a prime number p (here I have chosen a $Carol\ prime\ no.$ - 1046527) large enough so that every possible key k is in the range $0\ to\ p-1$, inclusive. We can now

define a hash function Hab for any a belonging to $\{1,2\,..\,p-1\}$ and b belonging to $\{0,1,\ldots,p-1\}$ using a linear transformation followed by reductions $modulo\ p$ and then $modulo\ m$: -

$$Hab(k) = ((ak+b)\ mod\ p)\ mod\ m$$

Since we have $p-1$ choices for a and p choices for b, the collection Hab contains $p(p-1)$ hash functions.

Theorem – $The\ class\ Hab\ of\ hash\ functions\ defined\ above\ is\ Universal.$

More formal proof is present in the text book, but according to the scope of my software - *Gene,* the probability of s and r colliding when reduced modulo m is at most $((p-1)/m)/(p-1) = 1/m$. Therefore, $Pr\{Hab\ (k) = Hab\ (l)\} <= 1/m$. Hence Hab is indeed Universal.

Conclusion – *Gene* is using the best approaches possible to satisfy both the research needs and conducting efficient logistics at Ribo Biotechs. Command line interface is made intuitive and user don't need to consult README file in detail.

****

References: -

1. *COMP3600 all lecture slides - https://cs.anu.edu.au/courses/comp3600/schedule.html*
2. *CLRS chapter 15.4 – [Dynamic Programming]*
3. *CLRS chapter 11 – [Hash Tables]*
4. *CLRS chapter 6 – [Heaps]*

Note – *Ribo Biotechs Pvt. Ltd.* Is a fictitious company.