Informe Proyecto 2 Analisis de Algoritmos

Integrantes: Linna Cao Merino

Jesús Gómez Zambrano

Bastián Gómez Jara

1- Bucket sort

El algoritmo de Bucket sort separa el arreglo inicial de datos no ordenados en distintos grupos llamados buckets, para ordenar este arreglo inicial, se hace un ordenamiento conveniente sobre cada bucket, para luego juntar los buckets ordenados, para entregar el arreglo inicial de datos ordenados.

Un Ejemplo de Bucket sort podría ser el ordenar un grupo de números enteros en el intervalo del 0 al 100.

Para el análisis de tiempo esperado podemos decir que como el algoritmo que implementamos, para el resolver el ejemplo dado, contiene dentro del ordenamientos de los buckets el algoritmo de sort, y el algoritmo implementado recorre a lo más los 10 vectores n veces, podemos concluir que nuestro peor tiempo será de O(nlogn) y el tiempo esperado para una distribución uniforme de los datos será de O(n).

2-Perfect Hashing

a) Descripcion estructura usada

En nuestro código para perfect hashing usamos una estructura de datos la cual llamamos “tabla” esta contiene tres variables enteras que son usadas para obtener y manejar colisiones y un vector de strings vec, que es donde se guardan los kmers de tamaño 15 del genoma introducido, la estructura también cuenta con 2 funciones internas una siendo el constructor de la tabla y la otra la función llamada “darTamano” que modifica el tamaño del vector vec de la estructura.

b) Pseudocodigos

```
- cases(c)
cod → 0
switch(c)
case 'G'
cod → 0
case 'T'
cod → 1
case 'C'
cod → 2
case 'A'
cod → 3
default cod → -1
return cod
- hlv1(k, aj, bj, m)
        ak ← aj*k
        h ← (((ak + bj )% p) % m)
return abs(h)
```

```
- kmer()
        kmers
        infile
        infile.open("input.txt")
        if(infile.fail())  print “error message”
        else
                s
                while(getline(infile, s)
                l ← s.length()-15
                if(l>0)
                        for i 0 to 1
                        do
                                k
                                cont ← 0
                                copy_n(s.begin()+i, 15, back_inserter(k))
                                kmers.insert(k)
        return kmers
- codificar(s)
        cod ← cases(s[0])
        if (cod<0) return INT_MIN
        for i 1 to 15
                do cod ← (cod<<2)| cases(s[i])
return cod
- DoHash1(s, a, b)
v(s.size())
for it s.begin() to s.end()
do v[hlv1(codificar(*it),a,b,s.size()-1)].col ← v[hlv1(codificar(*it),a,b,s.size()-1)].col +1
return  v
- getAB(s, n, colisiones)
        colisiones ← INT_MAX
        cont ← 0
        Final
        while(colisiones>n*s.size())
                colisiones ← 0
                a ← 1+rand()%p
                b ← 1+rand()%p
                v ← DoHash1(s,a,b)
                for it v.begin() to v.end
                do
                        aux ← it->col
                        aux ← aux * aux
                        colisiones ← colisiones + aux
                        if(colisiones<= n * s.size()) final = v
                        cont ← cont + 1
return final
- getAiBi(v)
        contbucket ← 0
        for it v.begin() to v.end()
        do
                if(it->col>1)
```

```
                colisiones ← 1
                vaux ← it->vec
                it->darTamano(pow(it->col,2))
                while(colisiones>0)
                        fill(it->vec.begin(),it->vec.end(),"%")
                        colisiones ← 0
                        it->ai ← rand()%p
                        it->bi ← rand()%p
                        for iter vaux.begin() to vaux.end()
                        do
                                hashpos ← hlv1(codificar(*iter),it->ai,it->bi,it->vec.size())
                                if(it->vec[hashpos]!="%")
                                        colisiones ← 1
                                        break
                                else
                                do it->vec[hashpos] ← it->vec[hashpos] * iter
- find(v, s, n)
        aux ← hlv1(codificar(s),a,b,n-1)
        if(v[aux].vec.size()==1)
        if(v[aux].vec[0]==s) printf found message
        else if(v[aux].vec.size()>1)
        if(v[aux].vec[hlv1(codificar(s),v[aux].ai,v[aux].bi,v[aux].vec.size())]==s)
        printf found message
        else
        do print not found message
- assignBuckets(s, v)
        for it s.begin() to s.end()
        do
                aux ← hlv1(codificar(*it),a,b,s.size()-1)
        if(v[aux].col>=1) v[aux].vec.push_back(*it)
        getAiBi(v)
```

c) Análisis de número esperado de colisiones

El promedio de colisiones totales para la tabla de primer nivel es de: 2.344.501, para las tablas de segundo nivel no existen colisiones debido a que la funcion getAiBi() se asegura de obtener un par de enteros Ai, Bi para cada bucket que garanticen eso.

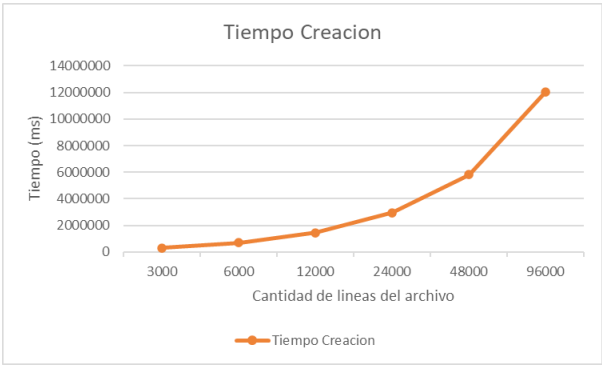
d) Análisis de complejidad de espacio esperado:

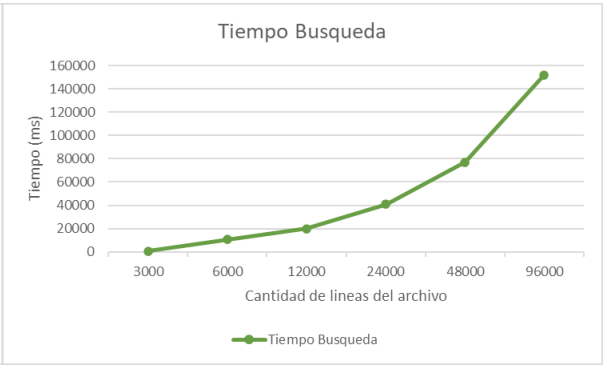
Sabemos por la forma en que definimos nuestras funciones que obligatoriamente la sumatoria del cuadrado de las colisiones va a ser menor a 4N (para una cota 4)*, por lo que el tamaño va a estar dado por la estructura tabla, cada tabla actúa realmente como un vector de elementos y posee su propio A y B además de su cantidad de colisiones. Considerando todo esto sabemos que la sumatoria de los vectores de todas las tablas es menor a 4N, además de este vector, ya que cada tabla almacena 3 enteros y son N tablas se tendrá 3N adicional, por lo que la complejidad de espacio esta dada por O(n). Saber el resultado del análisis de tiempo no nos ayudará en este caso porque puede que el código se ejecute muy rápido pero aun así use una gran cantidad de espacio.

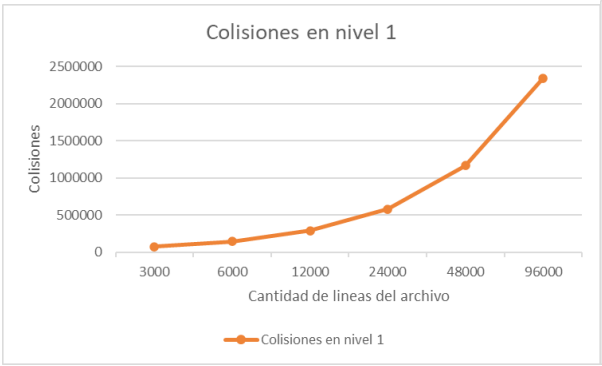
e) Gráficos análisis experimental:

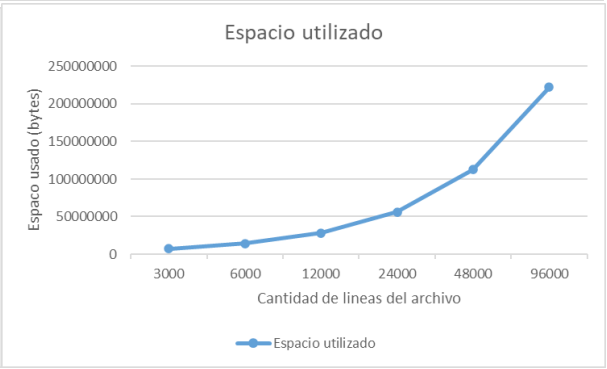
Se realizaron los análisis solicitados aumentando de manera logarítmica la entrada para obtener un gráfico más claro, se usó desde 3000 líneas hasta el total de líneas del archivo, duplicando la cantidad cada vez, si se desea saber la cantidad de k-mers, basta con multiplicar la cantidad de líneas por 66 pues cada línea tiene aproximadamente 66 k-mers, obteniendo los siguientes gráficos:

Tiempo de creación de la estructura









Podemos observar que tras el análisis experimental todos estos gráficos poseen un crecimiento lineal (Exponencial dentro de un gráfico logarítmico), por lo que podemos deducir que las funciones evaluadas son O(n) al igual que el numero de colisiones y las veces que se necesita repetir en total la búsqueda de un A, B apropiados para asegurar 0 colisiones en el segundo nivel.

Por su parte las veces que se necesita repetir la función que obtiene un A, B válido para la cota (se evaluó con una cota de 4N) se mantuvo constante en 1, en comparación con las de segundo nivel las cuales se tendrán que realizar en promedio n/3 veces, con n igual a la cantidad de k-mers de la estructura.

Ejemplo de input/output:

```
jesus@DESKTOP-5IBIE78:/mnt/c/Users/Jesus/Documents/1 - 2022/Analisis/p2$ ./main
inserte la cota de espacio de los buckets por favor
4
inserte el genoma a buscar o escriba '0' si desea terminar el programa
TGCGTGAAGAATTTC
Buscando el elemento TGCGTGAAGAATTTC
Encontrado
inserte el genoma a buscar o escriba '0' si desea terminar el programa
TGCGTGAAGAATTCC
Buscando el elemento TGCGTGAAGAATTCC
No encontrado
inserte el genoma a buscar o escriba '0' si desea terminar el programa
0
```

*No se logró obtener un A,B para la cota 2N, el código se ejecutaba de manera indefinida y nunca lograba ser menor a 2N

f) Discusión

De los resultados obtenidos durante este proyecto pudimos observar que a pesar de que nuestros inputs fueran del tamaño de millones de entradas, al usar perfect hashing en la búsqueda de elementos, el tiempo mantenía en complejidad lineal, por lo que podemos decir que es una forma optima de búsqueda de elementos, a su vez a el espacio utilizado en para esta búsqueda y la creación de las tablas también poseen un crecimiento lineal, siendo así bastante eficiente es el uso de espacio de igual manera.