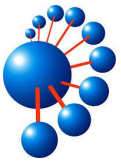


Proyecto 2:
Algoritmos aleatorizados

Análisis de Algoritmos (2022-1)

Integrantes: Dazhi Enrique Feng Zong
Pablo Ignacio Zapata Schifferli
Profesora: Cecilia Hernández R.
Fecha: 6 de junio, 2022

Bucket Sort

Algoritmo de ordenamiento que funciona distribuyendo los elementos del arreglo a un número finito de “buckets”. Después, cada bucket es ordenado con otro algoritmo de ordenamiento o con el mismo bucket sort (recursivamente). Finalmente, se juntan los buckets, para tener el arreglo ordenado. Entonces, los pasos son:

1. Crear buckets vacíos.
2. Colocar cada elemento del arreglo en algún bucket.
3. Ordenar buckets no vacíos.
4. Devolver los elementos de cada bucket concatenados por orden.

Pseudocódigo

```
Function bucketSort(array, k) is
      buckets ← new array of k empty lists
      for i = 0 to length(array) do
            c ← find bucket
            insert array[i] into bucket[c]
      Endfor
      for i = 0 to k do
            sort(buckets[i])
      Endfor
      return the concatenation of buckets[0], ...., buckets[k]
end function
```

Ejemplo

Suponer:
- Un arreglo, arr[6] = {3, 8, 5, 9, 1, 6}
- k = 2

Una función para asignar cada elemento del arreglo a un bucket:
- elem > 5 ? bucket2 : bucket1

Entonces:
- El primer bucket tendría los elementos: 3, 5, 1
- El segundo bucket tendría los elementos: 8, 9, 6

Al ordenarlos:
- bucket1: 1, 3, 5
- bucket2: 6, 8, 9

Finalmente al concatenarlos, el arreglo queda como:
- arr[6] = {1, 3, 5, 6, 8, 9}

Análisis de complejidad en términos de tiempo esperado

Sean *k* buckets con n_i elementos cada uno, ordenar cada bucket con selection sort tiene complejidad $O(n_i^2)$, con un valor esperado $E(n_i^2)$. Sea $X_{i,j}$ una variable aleatoria donde un elemento j es insertado en un bucket i, 1 cuando es insertado, con probabilidad $\frac{1}{k}$, 0 en caso contrario.

$$E(n_i^2) = E([\sum_{j=1}^{n} X_{i,j}]^2) = E(\sum_{j=1}^{n} X_{i,j} \sum_{l=1}^{n} X_{i,l}) = E(\sum_{j=1}^{n}\sum_{l=1}^{n} X_{i,j}X_{i,l})$$

Sean los casos:

Misma variable: $j \neq l$, n veces

$E(X^2_{i,j}) = 1^2(\frac{1}{k}) + 0^2(1 - \frac{1}{k}) = \frac{1}{k}$

Distinta variable: $j = l$, $n^2 - n = n(n-1)$ veces

$E(X_{i,j}X_{i,l}) = E(X_{i,j})\,E(X_{i,l}) = (\frac{1}{k})(\frac{1}{k}) = (\frac{1}{k})^2$

$$E(n_i^2) = E(\sum_{j=1}^{n}\sum_{l=1}^{n} X_{i,j}X_{i,l}) = E(\sum_{j=1}^{n} X^2_{i,j} + \sum_{j=1}^{n}\sum_{l=1,\, j\neq l}^{n} X_{i,j}X_{i,l}) = \sum_{j=1}^{n} E(X^2_{i,j}) + \sum_{j=1}^{n}\sum_{l=1,\, j\neq l}^{n} E(X_{i,j}X_{i,l})$$

$$= [n][\frac{1}{k}] + [n(n-1)][(\frac{1}{k})^2] = \frac{n}{k} + \frac{n(n-1)}{k*k}$$

La complejidad esperada de bucket sort de da por:

$$O(\sum_{i=1}^{k} E(n_i^2)) = O([k][\frac{n}{k} + \frac{n(n-1)}{k*k}]) = O(n + \frac{n(n-1)}{k})$$

Con $k = O(n)$, $O(n + \frac{n(n-1)}{n}) \Rightarrow O(n + (n-1)) = O(n)$

Hashing Perfecto

Un hash perfecto es una solución para almacenar un conjunto de elementos conocidos previamente, con el inconveniente de no poder insertar otros. La construcción del mismo asegura que no existan colisiones entre dos elementos, garantizando un acceso a ellos en tiempo constante. Su funcionamiento consiste en una tabla de primer nivel que almacena tablas de segundo nivel, cada tabla con un hash propio. Las tablas de segundo nivel son las encargadas de almacenar las claves.

Estructuras de datos implementadas

Atributos privados:

1. vector<vector<int>> tabla: Tabla hash de 2 niveles.
2. int k: Largo del k-mer, en este caso 15.
3. int rep: Variable que cuenta las veces requeridas para encontrar a y b.
4. int cota: Cota para la suma de cuadrados del tamaño de los buckets.
5. int a: Constante aleatoria de la función hash h().
6. int b: Constante aleatoria de la función hash h().
7. int m: Cantidad de k-mers distintos (número de buckets de la tabla).
8. int ai: Constante aleatoria de la función hash hi().
9. int bi: Constante aleatoria de la función hash hi().
10. int mi: $c_i^{\,2}$. Constante de la función hash hi().
11. int p: Siguiente primo mayor a 10m.

Métodos privados:

1. int h(int kmerc): Hash usado para tabla de primer nivel.
2. int hi(int kmerc): Hash usado para tabla de segundo nivel.
3. int nextPrime(int n): Próximo primo después de n.
4. int codificar(string kmer): Pasar el k-mer de string a int.
5. int procesarkmers(string &genoma, unordered_set<int> &setKmers): Hacer la división del genoma en k-mers y codificarlos.
6. void crearTabla(unordered_set<int> &setKmers): Crear la tabla hash, haciendo hash a los k-mers procesados.

Métodos públicos:

1. HashPerfecto(string &genoma): Constructor, inicializa las variables para procesar los k-mers y crear la tabla hash.
2. bool search(string kmer): Busca el k-mer en la tabla hash.
3. int repeticiones(): Devolver el número de repeticiones, para encontrar a y b.
4. int memoria(): Calcular la memoria utilizada.

Pseudocódigo de los métodos

Públicos:

```
HashPerfecto(string &genoma)
      k ← 15
      if length of genoma < k then
          throw invalid_argument("Genoma de largo insuficiente")
      unordered_set<int> setKmers
      m ← procesarkmers(genoma, setKmers)
      p ← nextPrime(10*m)
      rep ← 1
      cota ← 4*m
      Call crearTabla(setKmers)
```

```
search(string kmer)
      if length of kmer ≠ k then return False
      kmerc ← codificar(kmer)
      pos1 ← h(kmerc)
      if tabla[pos1]is empty then return False
      mi ← tabla[pos1][0]
      ai ← tabla[pos1][1]
      bi ← tabla[pos1][2]
      kmerTabla ← tabla[pos1][3 + hi(kmerc)]
      if kmerTabla  = kmerc then return True
      else return False

memoria()
      mem ← 0
      mem ← mem + sizeof(HashPerfecto)
      mem ← mem + sizeof(int)*10
      mem ← mem + sizeof(vector<vector<int>>)
      foreach vector<int> in tabla do
                  mem    ←    mem    +    sizeof(vector<int>)    +
sizeof(int)*v.capacity()
      return mem

repeticiones()
      return rep
```

Privados:

```
codificar(string kmer)
      kmerc ← 0
      for i = 0 to k do
            if kmer[i] = 'A' then
                kmerc += 0 << i*2
            else if kmer[i] = 'C' then
                kmerc += 1 << i*2
            else if kmer[i] = 'T' then
                kmerc += 2 << i*2
            else if kmer[i] = 'G' then
                kmerc += 3 << i*2
            else throw invalid_argument("Nucleotido invalido")
      return kmerc

nextPrime(int n)
      while true do
            flag ← 1
            n++
            if n <= 1 then continue
            else if n <= 3 then break
            else if n % 2 = 0 or n % 3 = 0 then continue
            else
                for i = 5, i*i<=n, i=i+6 do
                       if n % i = 0 or n % (i+2) = 0 then
                             flag ← 0
                             break
            if flag > 0 then break
      return n
```

```
h(int kmerc)
     return absolute value of (((a * kmerc + b) % p) % m)

hi(int kmerc)
     return absolute value of (((ai * kmerc + bi) % p) % mi)

procesarkmers(string &genoma, unordered_set<int> &setKmers)
     for i = 0 to length of genoma -(k-1) do
          kmer ← substring of genoma with length k
          kmerc ← codificar(kmer)
          insert kmerc into setKmers
     return the size of setKmers

crearTabla(unordered_set<int> &setKmers)
     Call minstd_rand rng(time(NULL))
     while true do
          a ← random number less than p
          b ← random number less than p
          vector<vector<int>> tabla1(m)
          for each int kmer in setKmers do
              pos ← get the position with hashing, h(kmer)
              insert kmer into tabla1[pos]
          c ← 0
          for each vector<int> v in tabla1 do
              c ← c + v.size()*v.size();
          if c < cota then
               tabla ← tabla1
               break
          else rep ← rep + 1
     for i = 0 to m do
          mi ← tabla[i].size() * tabla[i].size()
          count ← 0
          while mi > 0 do
               colision ← false
               ai ← random number less than p
               bi ← random number less than p
               vector<int> tabla2(3 + mi)
               for each int kmer in tabla[i] do
                    pos ← 3 + hi(kmer)
                    if tabla2[pos] = 0 then tabla2[pos] = kmer
                    else
                        colision ← true
                        break
               if colision = false then
                    tabla2[0] ← mi
                    tabla2[1] ← ai
                    tabla2[2] ← bi
                    tabla[i] ← tabla2
                    break
```

Análisis de colisiones esperadas

Por cada tabla de segundo nivel, de tamaño m_i, existen tienen c_i claves con una probabilidad de colisión entre dos de ellas igual a $\frac{1}{m_i}$. Pueden haber $(c_i\, nCr\, 2)$ pares de claves que pueden colisionar.

$(c_i\, nCr\, 2) = \frac{c_i!}{2!(c_i-2)!} = c_i(c_i - 1)\frac{1}{2}$

X = variable aleatoria de número de colisiones

$E(X) = c_i(c_i - 1)\frac{1}{2} \cdot \frac{1}{m_i} = c_i(c_i - 1)\frac{1}{2} \cdot \frac{1}{c_i \cdot c_i} = (1 - c_i^{-1})\frac{1}{2} < \frac{1}{2}$

si $m_i = c_i \Rightarrow E(X) = (c_i - 1)\frac{1}{2}$

Como se observa, al hacer $m_i = c_i^2$ el número esperado de colisiones es menor a $\frac{1}{2}$, al no hacer este paso, usando $m_i = c_i$, las colisiones esperadas están en relación al número de claves: c_i.

Análisis de complejidad de espacio esperado

Considerando el análisis de complejidad del bucket sort, en vez de k buckets con n_i elementos, en el hash perfecto son $k = n$ tablas de segundo nivel con $n_i = c_i$ elementos. En bucket sort cada bucket tiene una complejidad esperada $E(n_i^2)$, causado por usar selection sort, aquí debido a que hacemos $m_i = c_i^2$, el espacio esperado es $E(c_i^2)$. Finalmente el desarrollo es el mismo al de bucket sort:

$X_{i,j}$: variable aleatoria donde un elemento j es puesto en una tabla de segundo nivel i, 1 cuando es insertado en la tabla, con probabilidad $\frac{1}{n}$, 0 en caso contrario.

$E(c_i^2) = E([\sum_{j=1}^{n} X_{i,j}]^2) = E(\sum_{j=1}^{n} X_{i,j} \sum_{l=1}^{n} X_{i,l}) = E(\sum_{j=1}^{n}\sum_{l=1}^{n} X_{i,j}X_{i,l})$

Sean los casos:

Misma variable: $j \neq l,\ n\ veces$

$E(X^2_{i,j}) = 1^2(\frac{1}{n}) + 0^2(1 - \frac{1}{n}) = \frac{1}{n}$

Distinta variable: $j = l,\ n^2 - n$ veces

$E(X_{i,j}X_{i,l}) = (\frac{1}{n})(\frac{1}{n}) = (\frac{1}{n})^2$

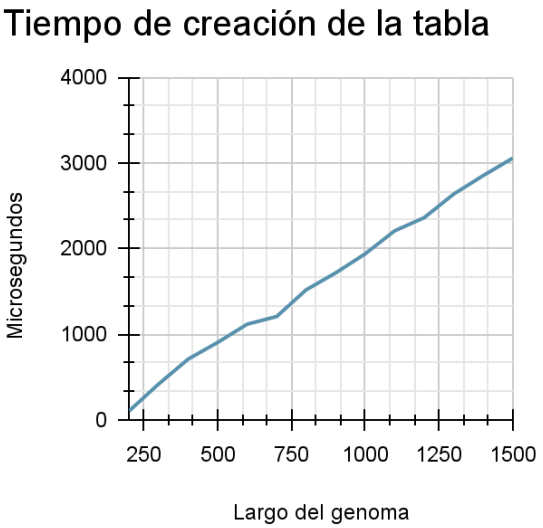
$E(\sum_{j=1}^{n}\sum_{l=1}^{n} X_{i,j}X_{i,l}) = \sum_{j=1}^{n} E(X^2_{i,j}) + \sum_{j=1}^{n}\sum_{l=1, j\neq l}^{n} E(X_{i,j}X_{i,l}) = [n][\frac{1}{n}] + [n^2 - n][(\frac{1}{n})^2] = 2 - \frac{1}{n}$

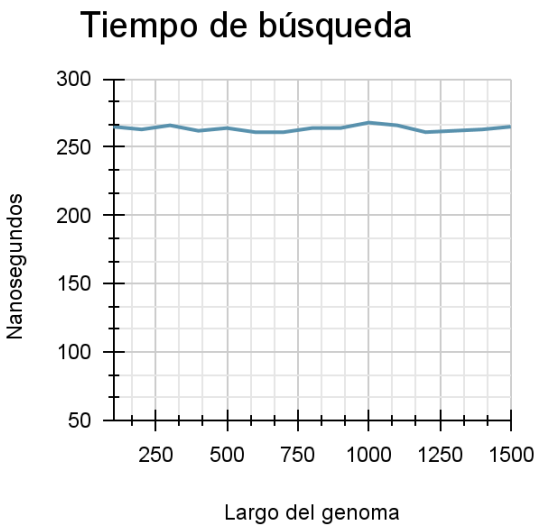
$E(c_i^2) = 2 - \frac{1}{n}$

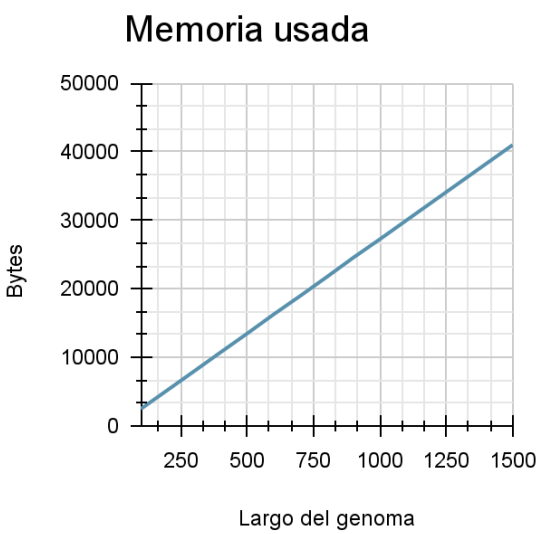
Complejidad espacial del Hash Perfecto:

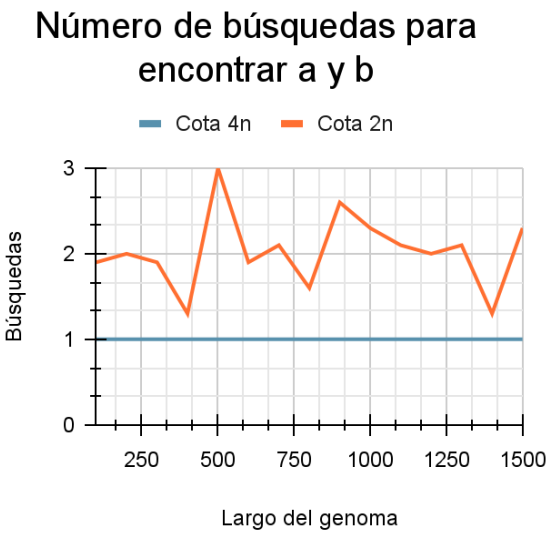
$O(\sum_{i=1}^{n} E(c_i^2)) = O([n][2 - \frac{1}{n}]) = O(2n - 1) = O(n)$

Evaluación experimental









Del analisis teorico y experimental, se puede ver que:

- La creación de la tabla toma un tiempo lineal, ya que tiene que hacerle hashing a todos los elementos (k-mers). Como son m elementos y el hash es constante, la creación de la tabla toma tiempo $O(m)$.
- Se sabe que el tiempo que toma para buscar un elemento en una tabla hash generalmente es constante y se puede ver con el gráfico que esto es verdadero.
- Se puede intuir que una cota más pequeña (2n) para encontrar los valores a y b, va tomar más tiempo y una cota más grande (4n) va tomar menos. Esto también se comprueba con el análisis experimental.
- Del análisis teórico, sabemos que el espacio esperado es lineal y del análisis experimental, lo podemos comprobar.