ANÁLISIS TEÓRICO

- Find:

  - *Primer método find*

```
pair<conjunto::value_type,bool>  conjunto::find (const conjunto::value_type & e){

   conjunto::iterator it;

   int Iarriba = vm.size()- 1;                    // O(1)
   int Iabajo = 0;                                // O(1)
   int Icentro;

   while (Iabajo <= Iarriba)                      // Comparación:  0(1)
   {
      Icentro = (Iarriba + Iabajo)/2;             // O(1)

      if (this->vm.at(Icentro) == e){             // Condición: O(1)
         it = vm.begin() + Icentro;               // O(1)
         return pair<value_type,bool>(*it,true);
      }

      else if (e < this→vm.at(Icentro))           // Condición: O(1)
         Iarriba=Icentro-1;                       // O(1)

      else
         Iabajo=Icentro+1;                        // Condición: O(1)
   }

   return pair<value_type,bool>(mutacion(),false);
}
```

Como todas las sentencias son de O(1), el tiempo del bucle será la suma de O(1) en el número de iteraciones que se de. El peor de los casos sería que un elemento no se encontrara en el conjunto o que ese elemento se encontrara en la última comparación.

$$\sum_{n=\text{tamaño del conjunto}}^{\log_2(n)} O(1) = \mathbf{O(\log_2(n))}$$

En cada iteración que falla, se disminuye la longitud del subconjunto que queda por un factor de 2. El tamano de los subconjuntos es:

n $\quad \frac{n}{2} \quad \frac{n}{2^2} \quad \frac{n}{2^3} \quad \dots \quad 1$

La última iteración sería $\frac{n}{2^m} = 1$, siendo m el número de iteraciones.

- *Segundo método find*

```
pair<conjunto::value_type,bool>  conjunto::find (const string & ID){

   conjunto::iterator it;

   for(it = vm.begin() ; it != vm.end(); ++it){          // Comparación: O(1) ; Incremento: O(1)
      if( it->getID() == ID )                            // Comparación: O(1)
         return pair<conjunto::value_type,bool>(*it,true);
   }

   return pair<conjunto::value_type,bool>(mutacion(),false);
}
```

El método se compone de un bucle for que recorre todo el conjunto. Como las sentencias son de O(1), el tiempo será la suma de O(1) en el número de iteraciones que se de.

$$\sum_{it=0}^{n} O(1) = \mathbf{O(n)}$$

En el peor de los casos, el elemento buscado no estará en el conjunto o estará en la última posición del conjunto, es decir, hará tantas iteraciones como elementos contenga el conjunto.

- *Tercer método find*

```
pair<conjunto::value_type,bool>  conjunto::find (const string & chr, const unsigned int & pos){

   value_type m_aux;
   m_aux.setChr(chr);                    // O(1)
   m_aux.setPos(pos);                    // O(1)
   return find(m_aux);                   // O(log2(n))

}
```

Este método crea una mutación con el cromosoma y la posición correspondiente y llama al método *find* que antes hemos analizado, obteniendo $O(\log_2(n))$. Con lo cual, aplicamos la regla de la suma. El tiempo del método completo será el $\max(O(1), O(1), O(\log_2(n)))$. El maximo es $\mathbf{O(\log_2(n))}$.

- Insert

```
bool conjunto::insert(const conjunto::value_type & e){

   if(chr_correcto(e)) {                                          // O(?)
      pair<conjunto::value_type,bool> e_aux = find(e); // O(log2(n))
      conjunto::iterator it;

      if (e_aux.second == false){                                 // O(1)
         it = lower_bound(e);                                     // O(?)

         if(it == vm.end()){                                      // O(1)
            vm.push_back(e);                                      // O(1) (según cplusplus.com)
            return true;
         }

         else{
            vm.insert(it, e);                                     // O(n) (según cplusplus.com)
            return true;
         }
      }
   }

   return false;
}
```

Para analizar este método, tenemos que analizar la eficiencia de los métodos *chr_correcto* y *lower_bound*

```
conjunto::iterator conjunto::lower_bound (const conjunto::value_type & e){

   conjunto::iterator it, first, last;

   first = vm.begin();
   last = vm.end();

   typename iterator_traits<conjunto::iterator>::difference_type contador, step;
   contador = distance(first, last);

   while (contador > 0) {                    // Comparación: O(1)
      it = first;                            // O(1)
      step = contador / 2;                   // O(1)
      advance(it, step);                     // O(1)
      if (*it < e) {                         // Comparación: O(1)
         first = ++it;                       // O(1)
         contador -= step + 1;               // O(1)
      } else contador = step;                // O(1)
   }

   return first;
}
```

Este es un caso parecido al método *find*. Las sentencias que contiene son de O(1), en cada iteración que falla, se disminuye la longitud del subconjunto que queda por un factor de 2. La diferencia es que en este método, si un elemento no se encuentra devuelve el elemento que se considera mayor que el elemento buscado. El peor de los casos sería que el elemento no se encontrara y devolviera la primera posición del conjunto o la última, es decir, que el bucle llega hasta la última comparación.

$$\sum_{n=tamaño\,del\,conjunto}^{\log_2(n)} O(1) = O(\log_2(n))$$

```
bool chr_correcto(const conjunto::value_type & e){
   string chr_aux;
   int chr_num;

   if(e.getChr() == "")                                    // O(1)
      return false;

   if(e.getChr().compare("X") == 0)                // O(1)
      chr_aux = "23";                                      // O(1)
   else if(e.getChr().compare("Y") == 0)           // O(1)
      chr_aux = "24";                                      // O(1)
   else if(e.getChr().compare("MT") == 0)          // O(1)
      chr_aux = "25";                                      // O(1)
   else
      chr_aux = e.getChr();                                // O(1)

   chr_num = stoi(chr_aux);                        // O(1)

   return (1 <= chr_num && chr_num <= 25 && e.getPos() > 0);                  // O(1)
}
```

Este método tiene sentencias simples de comparación y asignación, todas de O(1), tanto el bloque *if* como el bloque *else*, entonces el método completo tiene O(1).

Ahora ya podemos analizar el método insert completo.
El bloque *if/else* más interno tiene O(n) debido al método insert en el bloque *else*. Ese bloque está dentro de un bloque *if* que, aplicando la regla de la suma, tiene O(n) ya que contiene el *if/else* anterior. Ese bloque *if* está dentro de un último bloque *if*, con la condición del método *chr_correcto* que tiene O(1) y que, aplicando la regla de la suma en todo el cuerpo obtenemos un **O(n).**

- *Erase*
  - *Primer método erase*

```
bool conjunto::erase(const conjunto::value_type & e){
   pair<conjunto::value_type,bool> e_aux = find(e);        // O(log2(n))
   conjunto::iterator it;

   if(e_aux.second == true){                               // Comparación: O(1)
      it = lower_bound(e);                                 // O(log2(n))
      vm.erase(it);                                        // O(n)
      return true;
   }

   return false;
}
```

A este método aplicamos la regla de la suma entre el bloque *if* y el método *find.* El *if* tiene mayor orden, con lo cual el método completo tiene **O(n).**

  - *Segundo método erase*

```
bool conjunto::erase(const string & ID){
   conjunto::iterator it;

   for(it = vm.begin() ; it != vm.end(); ++it){
      if( it->getID() == ID ){                             // Comparación: O(1)
         vm.erase(it);                                     // O(n)
         return true;
      }
   }

   return false;
}
```

Este método está compuesto de un bucle *for* que recorre todos los elementos del conjunto. El cuerpo del bucle se compone de un *if* que tiene O(n) debido al método *erase*.

$$\sum_{it=1}^{n} O(n) \quad = \mathbf{O(n^2)}$$

  - *Tercer método erase*

```
bool conjunto::erase(const string & chr, const unsigned int & pos){
   value_type e_aux;
   e_aux.setChr(chr);                  // O(1)
   e_aux.setPos(pos);                  // O(1)
   return erase(e_aux);                // O(n)
}
```

Este método llama al erase anterior analizado. Aplicamos la regla de la suma, obteniendo un **O(n)**.