
Algoritmo 1: División Greedy

```
Var Global: ispellList // Palabras de ispell + Diccionario
Var Global: abrevList // Abreviaciones conocidas
Var Global: stopList // Palabras Excluyentes
Entrada  : idHarword // identificador a dividir
Salida   : softwordDiv // id separado con espacios

1 softwordDiv  $\leftarrow$  ""
2 softwordDiv  $\leftarrow$  dividirCaracteresEspecialesDigitos(idHarword)
3 softwordDiv  $\leftarrow$  dividirCamelCase(softwordDiv)
4 para todo (s | s es un substring separado por ' ' en softwordDiv)
  hacer
5   si (s no pertenece a (stopList  $\cup$  abrevList  $\cup$  ispellList ))
     entonces
6       resultadoPrefijo  $\leftarrow$  buscarPrefijo(s, "")
7       resultadoSufijo  $\leftarrow$  buscarSufijo(s, "")
           // Se elije la división que mayor particiones hizo.
8       s  $\leftarrow$  maxDivisión(resultadoPrefijo, resultadoSufijo)

9 devolver softwordDiv // Retorna el id dividido por ' '
```

Función buscarPrefjjo

Entrada: s // Abreviaturas a dividir

Salida : *abrevSeparada* // Abreviaturas separadas

// Punto de parada de la recursión.

1 si ($length(s) = 0$) entonces

2 devolver *abrevSeparada*

3 si (s pertenece a ($stopList \cup abrevList \cup ispellList$)) entonces

4 devolver ($s + ' ' + buscarPrefjjo(abrevSeparada, "")$)

// Se extrae y se guarda el último caracter de s .

5 $abrevSeparada \leftarrow s[length(s) - 1] + abrevSeparada$

// Llamar nuevamente a la función sin el último caracter.

6 $s \leftarrow s[0, length(s) - 1]$

7 devolver $buscarPrefjjo(s, abrevSeparada)$

Función buscarSufijo

Entrada: s // Abreviaturas a dividir

Salida : *abrevSeparada* // Abreviaturas separadas

// Punto de parada de la recursión.

1 si ($length(s) = 0$) entonces

2 devolver *abrevSeparada*

3 si (s pertenece a ($stopList \cup abrevList \cup ispellList$)) entonces

4 devolver ($buscarSufijo(abrevSeparada, "") + ' ' + s$)

// Se extrae y se guarda el primer caracter de s .

5 $abrevSeparada \leftarrow abrevSeparada + s[0]$

// Llamar nuevamente a la función sin el primer caracter.

6 $s \leftarrow s[1, length(s)]$

7 devolver $buscarSufijo(s, abrevSeparada)$

Algoritmo 2: divisiónHardWord

Entrada: *token* // *token a dividir*

Salida : *tokenSep* // *token separado con espacios*

```
1 token ← dividirCaracteresEspecialesDigitos(token)
2 token ← dividirMinusSeguidoMayus(token)
3 tokenSep ← ""
4 para todo (s | s es un substring separado por ' ' en token) hacer
5     si (  $\exists \{i | esMayus(s[i]) \wedge esMinus(s[i + 1])\}$  ) entonces
6         n ← length(s) - 1
7         // se determina con la función score si es del tipo
8         // camelcase u otra alternativa
9         scoreCamel ← score(s[i,n])
10        scoreAlter ← score(s[i+1,n])
11        si (scoreCamel >  $\sqrt{scoreAlter}$ ) entonces
12            si (i > 0) entonces
13                s ← s[0,i - 1] + ' ' + s[i,n] // GP Sstate
14            en otro caso
15                s ← s[0,i] + ' ' + s[i + 1,n] // GPS state
16        tokenSep ← tokenSep + ' ' + s
17 token ← tokenSep
18 tokenSep ← ' '
19 para todo (s | s es un substring separado por ' ' en token) hacer
20     tokenSep ← tokenSep + ' ' + divisiónSoftWord(s,score(s))
21 devolver tokenSep
```

Algoritmo 3: divisiónSoftWord

Entrada: s // *softword string*

Entrada: $score_{sd}$ // *puntaje de s sin dividir*

Salida : $tokenSep$ // *token separado con espacios*

```
1  $tokenSep \leftarrow s$ ,  $n \leftarrow \text{length}(s) - 1$ 
2  $i \leftarrow 0$ ,  $maxScore \leftarrow -1$ 
3 mientras ( $i < n$ ) hacer
4      $score_{izq} \leftarrow \text{score}(s[0,i])$ 
5      $score_{der} \leftarrow \text{score}(s[i+1,n])$ 
6      $preSuf \leftarrow \text{esPrefijo}(s[0,i]) \vee \text{esSufijo}(s[i+1,n])$ 
7      $split_{izq} \leftarrow \sqrt{score_{izq}} > \max(\text{score}(s), score_{sd})$ 
8      $split_{der} \leftarrow \sqrt{score_{der}} > \max(\text{score}(s), score_{sd})$ 
9     si ( $\neg preSuf \wedge split_{izq} \wedge split_{der}$ ) entonces
10         si ( $(split_{izq} + split_{der}) > maxScore$ ) entonces
11              $maxScore \leftarrow (split_{izq} + split_{der})$ 
12              $tokenSep \leftarrow s[0,i] + ' ' + s[i+1,n]$ 
13         sinó, si ( $\neg preSuf \wedge split_{izq}$ ) entonces
14              $temp \leftarrow \text{divisiónSoftWord}(s[i+1,n], score_{sd})$ 
15             si ( $temp$  se dividió?) entonces
16                  $tokenSep \leftarrow s[0,i] + ' ' + temp$ 
17          $i \leftarrow i+1$ 
18 devolver  $tokenSep$ 
```

Algoritmo 4: Expansión Básica

Entrada: *abrev* // Abreviatura a expandir
Entrada: *listaPalabras* // Palabras extraídas del código
Entrada: *listaFrases* // Frases extraídas del código
Entrada: *stopList* // Palabras Excluyentes
Entrada: *dicc* // Diccionario en Inglés
Salida : *únicaExpansión* // Abreviatura expandida, o null

```
1 si (abrev pertenece stopList) entonces
2   └ devolver null

3 listaExpansión ← [ ]

   // Buscar coincidencia de acrónimo.
4 para todo (Frase | Frase es una frase en listaFrases) hacer
5   └ si (abrev es un acrónimo de Frase) entonces
6     └ // Se prioriza aquella Frase que está en el mismo
        método que abrev
        └ devolver Frase

   // Buscar abreviatura común.
7 para todo (Pal | Pal es una palabra en listaPalabras) hacer
8   └ si (abrev es una abreviatura de Pal) entonces
9     └ // Se prioriza aquella Pal que está en el mismo
        método que abrev
        └ devolver Pal

   // Si no hay éxito, buscar en el diccionario.
10 listaCandidatos ← buscarDiccionario(abrev, dicc)
    listaExpansión.add(listaCandidatos)

11 únicaExpansión ← null

   // Debe haber un solo resultado, sino no retorna nada.
12 si (length(listaExpansión) = 1) entonces
13   └ únicaExpansión ← listaExpansión[0]

14 devolver únicaExpansión
```

Algoritmo 5: Búsqueda por Palabras Singulares

Entrada: *pa* // *Palabra Abreviada*
Entrada: *patrón* // *Expresión regular*
Entrada: Cuerpo y Comentarios del Método
Entrada: Comentarios de la Clase
Salida : Palabras largas candidatas, o null si no hay

// Las expresiones regulares están entre comillas

1 **si** (*patrón* prefijo \vee *pa* coincide “[a-z][^aeiou]+” \vee length(*pa*) > 3)
 \wedge (*pa* no coincide con “[a-z][aeiou][aeiou]+”) **entonces**

*// Si alguna de las siguientes búsquedas encuentra un
 único resultado, el algoritmo lo retorna
 finalizando la ejecución*

2 Buscar en Comentarios JavaDoc con “@param *pa patrón*”

3 Buscar en Nombres de Tipos y la correspondiente Variable
 declarada con “*patrón pa*”

4 Buscar en el Nombre del Método con “*patrón*”

5 Buscar en las Sentencias con “*patrón pa*” y “*pa patrón*”

6 **si** (length(*pa*) \neq 2) **entonces**

7 Buscar en palabras del Método con “*patrón*”

8 Buscar en palabras que están en los Comentarios del Método
 con “*patrón*”

9 **si** (length(*pa*) > 1) \wedge (*patrón* prefijo) **entonces**

// Solo se busca con patrones prefijos

10 Buscar en palabras que están en los Comentarios de la Clase
 con “*patrón*”

Algoritmo 6: Búsqueda por Multi Palabras

Entrada: *pa* // Palabra Abreviada

Entrada: *patrón* // Expresión regular

Entrada: Cuerpo y Comentarios del Método

Entrada: Comentarios de la Clase

Salida : Palabras largas candidatas, o null si no hay

// Las expresiones regulares están entre comillas

1 **si** (*patrón* acrónimo \vee $\text{length}(\textit{pa}) > 3$) **entonces**

*// Si alguna de las siguientes búsquedas encuentra un
único resultado, el algoritmo lo retorna
finalizando la ejecución*

2 Buscar en Comentarios JavaDoc con “@param *pa patrón*”

3 Buscar en Nombres de Tipos y la correspondiente Variable
declarada con “*patrón pa*”

4 Buscar en el Nombre del Método con “*patrón*”

5 Buscar en todos los ids (y sus tipos) dentro del Método con
“*patrón*”

6 Buscar en Literales String con “*patrón*”

*// En este punto se buscó en todos los lugares
posibles dentro del método*

7 Buscar en palabras que están en los Comentarios del Método con
“*patrón*”

8 **si** (*patrón* acrónimo) **entonces**

// Solo se busca con patrones Acrónimos

9 Buscar en palabras que están en los Comentarios de la Clase
con “*patrón*”
