

# Distancia de Edición

Alejandro Pernin      Lautaro Medrano

18 de noviembre de 2013

# Índice

<b>1. Enunciado</b>	<b>3</b>
1.1. Distancia de edición . . . . .	4
1.2. Alineación secuencias ADN . . . . .	4
<b>2. Resolución</b>	<b>6</b>
2.1. Distancia de edición . . . . .	6
2.2. Alineación secuencias ADN . . . . .	7
<b>3. Análisis de orden</b>	<b>9</b>
<b>4. Ejecución y pruebas</b>	<b>10</b>
<b>5. Conclusiones</b>	<b>11</b>
<b>6. Reformas respecto 1º informe</b>	<b>12</b>
<b>7. Codigos Fuente</b>	<b>13</b>
7.1. LCS.py . . . . .	13
7.2. tdatp2.py . . . . .	14

# 1. Enunciado

**Distancia de edición** Para convertir una cadena de caracteres  $x[1..m]$  en otra cadena  $y[1..n]$ , se pueden realizar distintas operaciones. La meta consiste, en dadas las cadenas  $x$  e  $y$ , encontrar una serie de transformaciones para cambiar  $x$  en  $y$ , para lo cual debemos emplear un vector  $z$  (cuyo tamaño asumimos que es suficiente para almacenar todos los caracteres necesarios) en el que almacenaremos los resultados parciales. Al inicio  $z$  está vacío y al final se cumple que  $z[j] = y[j]$  para  $j=1, 2, \dots, n$ . Se deben examinar todos los caracteres de  $x$ , para lo cual se mantienen los índices  $i$  en  $x$  y  $j$  en  $z$ . En un principio  $i=j=1$  y al final  $i=m+1$ . En cada paso se aplica alguna de las siguientes seis operaciones (transformaciones):

**Copiar:** copia un carácter de  $x$  a  $z$ . Esto es:  $z[j] = x[i]$  e incrementa los índices  $i$  y  $j$

**Reemplazar:** reemplaza un carácter de  $x$  por otro carácter  $c$ . Esto es:  $z[j] = c$  e incrementa los índices  $i$  y  $j$

**Borrar:** borra un carácter de  $x$  incrementando  $i$  y sin mover  $j$

**Insertar:** inserta un carácter  $c$  en  $z$ . Esto es:  $z[j] = c$  e incrementa  $j$  sin mover  $i$

**Intercambiar:** intercambia los próximos dos caracteres copiándolos de  $x$  a  $z$  pero en orden inverso. Esto es:  $z[j] = x[i+1]$  y  $z[j+1] = x[i]$  e incrementa los índices de la siguiente manera:  $i=i+2$  y  $j=j+2$

**Terminar:** elimina los caracteres restantes de  $x$  haciendo  $i=m+1$ . Esta operación descarta todos los caracteres de  $x$  que todavía no se analizaron. Es la última operación se aplica si hace falta.

## 1.1. Distancia de edición

Dadas dos cadenas  $x$   $[1..m]$  e  $y$   $[1..n]$  y el costo de cada una de las operaciones. Escribir un programa (programación dinámica) que calcule la distancia de edición siendo esta la secuencia de menor costo que permita transformar  $x$  en  $y$ . Analizar la complejidad en tiempo y espacio de la solución implementada. El programa debe tomar como parámetros las dos cadenas y el nombre de un archivo con el costo de las operaciones. El formato de cada línea de este archivo es el siguiente:

`<operacion>:<costo>`

Ejemplo de invocación: *tdatp2 algoritmo altruista costos.txt*

La salida debe ser por pantalla y debe mostrar una línea por cada operación, indicando los caracteres que se insertan o reemplazan. Una línea en blanco y el costo total de la secuencia.

## 1.2. Alineación secuencias ADN

El problema de la distancia de edición, tal como está planteado en este TP generaliza el problema de alinear dos secuencias de ADN. Existen varios métodos para medir la similitud entre dos secuencias de ADN alineándolas. Uno de esos métodos consiste en insertar espacios en posiciones arbitrarias en ambas secuencias (incluyendo al final) tal que las secuencias resultantes tengan la misma longitud pero que no tengan espacios en la misma posición. Entonces se le asigna un puntaje a cada posición de la siguiente manera:

- +1 si  $x'[j] = y'[j]$  y ninguno es un espacio
- -1 si  $x'[j] \neq y'[j]$  y ninguno es un espacio
- -1 si  $x'[j]$  o  $y'[j]$  es un espacio

El puntaje total de la alineación es la suma de los puntaje de cada posición. Por ejemplo dadas las secuencias  $x=\text{GATCGGCAT}$  e  $y=\text{CAATGTGAATC}$ , una alineación posible es:

<i>G</i>		<i>A</i>	<i>T</i>	<i>C</i>	<i>G</i>		<i>G</i>	<i>C</i>	<i>A</i>	<i>T</i>	
<i>C</i>	<i>A</i>	<i>A</i>	<i>T</i>		<i>G</i>	<i>T</i>	<i>G</i>	<i>A</i>	<i>A</i>	<i>T</i>	<i>C</i>
-1	-2	+1	+1	-2	+1	-2	+1	-1	+1	+1	-2

Tal que el puntaje total de la alineación es  $6 * 1 - 2 * 1 - 4 * 2 = -4$   
Explicar que como utilizar el programa de distancia de edición del punto

1 utilizando un subconjunto de las operaciones copiar, reemplazar, borrar, insertar, intercambiar y terminar para resolver el problema de la alineación de secuencias de ADN con el método dado.

## 2. Resolución

### 2.1. Distancia de edición

Para la resolución de los problemas planteados se tuvieron en cuenta algunas hipótesis:

- El costo individual de copiar y reemplazar, son menores que los costos de las combinaciones de borrar e insertar.
- Conozco soluciones parciales anteriores del problema.
- El problema es case sensitive, por lo cual mayúsculas y minúsculas serán diferenciadas.
- Las operaciones están sujetas a prohibición.

El problema se atacó desde un punto de vista dinámico utilizando el enfoque de memorización, es decir que se van guardando en memoria soluciones previas a un problema menor, cuya solución considero es óptima. Para ello se utilizó un diccionario que irá alojando las soluciones previas para tenerlas disponibles para ensamblar soluciones futuras. El concepto de solución es:

- $S_n = S_{n-1} + s(n)$

Es decir que la solución hasta un carácter se compone de la solución hasta el carácter anterior más una solución para ese único carácter.

Para la composición de la solución se hace uso de la hipótesis de que la copia es la mejor operación a realizar (menor costo), para poder realizar una toma de decisiones se utiliza como recursos el resultado obtenido de una implementación de LCS<sup>1</sup> y un análisis de la diferencia de tamaños entre la base y el objetivo.

Algunas de las consideraciones que se toman en cuenta son:

- Si  $base(pos) \in LCS \rightarrow$  es un carácter que se puede copiar, si se puede copiar esa misma instancia u otra con sus respectivos impactos.
- Si  $|base| \leq |objetivo| \rightarrow$  borrar un carácter implica a futuro una inserción.

---

<sup>1</sup>Longest Common Sequence - Secuencia Común más Larga

- Si  $|base| \geq |objetivo| \rightarrow$  una inserción puede a futuro implicar un borrar o terminar.

Teniendo en cuenta estas consideraciones se hace un análisis de costos y de que operación es la óptima a efectuar.

```

Data: n
Result: lista_solucion
if  $post == 0$  then
  | return  $s(n)$ ;
end
if  $!Dict_{n-1}$  then
  |  $Dict_{n-1} = S_{n-1}$ ;
end
if  $!Dict_n$  then
  |  $Dict_{n-1} = Dict_{n-1} + s(n)$ ;
end
 $l \leftarrow Dict_n$ ;
return  $l$ ;

```

**Algorithm 1:** Obtención  $S_n$  con memorización

## 2.2. Alineación secuencias ADN

Para la resolución del problema de alinear secuencias de ADN se utiliza el mismo algoritmo utilizado para la distancia de edición, utilizando un conjunto de operaciones adaptados a este problema. Para hallar que operaciones y que respectivos costos se debieran utilizar se hizo un análisis del problema. Veamos el ejemplo de alineación:

	1	2	3	4
<i>base :</i>	G		A	C
<i>objetivo :</i>	C	A	A	
<i>costo :</i>	-1	-2	+1	-2

- 1) Representa un reemplazo con un costo de -1.
- 2) Representa una inserción con un costo de -2.
- 3) Representa un copiar con un costo de +1.

- 4) Representa un borrar con un costo de -2.
- No se realizan operaciones terminar o intercambiar.

El objetivo de la alineación es obtener aquella alineación cuyo puntaje es máximo, pero siendo que el algoritmo a utilizar trabaja con mínimos, lo que se debe realizar es invertir los costos (eg: Copiar:-1); así mismo el resultado que obtendremos también debemos invertirlo para que represente el puntaje máximo. Por ejemplo el archivo de costos a utilizar sería:

```
1 reemplazar:1
2 insertar:2
3 borrar:2
4 copiar:-1
```

Utilizando este archivo para la resolución de las secuencias dadas como ejemplo obtenemos:

```
1 reemplazar g c
2 copiar a
3 insertar a
4 copiar t
5 reemplazar c g
6 reemplazar g t
7 copiar g
8 reemplazar c a
9 copiar a
10 copiar t
11 insertar c
12
13 El costo es: 3
```

Cuyo puntaje de alineación es -3 y equivale a:

<i>G</i>		<i>A</i>	<i>T</i>	<i>C</i>	<i>G</i>	<i>G</i>	<i>C</i>	<i>A</i>	<i>T</i>	
<i>C</i>	<i>A</i>	<i>A</i>	<i>T</i>	<i>G</i>	<i>T</i>	<i>G</i>	<i>A</i>	<i>A</i>	<i>T</i>	<i>C</i>
-1	-2	+1	+1	-1	-1	+1	-1	+1	+1	-2



### 3. Análisis de orden

A continuación se hará un análisis de los órdenes de los principales módulos que se utilizan en la resolución del problema. Se omiten aquellos que no revisten complejidad y/o relevancia, pero cuyo análisis igual se encuentra en el código fuente<sup>2</sup>.

- LCS<sup>3</sup>: El algoritmo consta de tres segmentos, el costo total es la suma de los costos individuales.

$$O(|base| + 1 * |obj| + 1) + O(|base| * |obj|) + O(|base| * |obj|)$$

$$O(|base| + 1 * |obj| + 1) = O(|base| * |obj| + |base| + |obj| + 1) < O(3 * |base| * |obj|) = O(|base| * |obj|)$$

Por propiedades de O, el orden total es:

$$O(|base| * |obj|)$$

- solve()<sup>4</sup>: Todas las operaciones realizadas son  $O(1)$ , hay un llamado recursivo, pero de profundidad 1; por lo que se puede asegurar que el costo total de este módulo es  $O(1)$
- solution(): Por cada elemento de la palabra objetivo, se analiza si existe o no solución en tiempo constante.  $O(|obj|)$
- La solución total se compone de efectuar un llamado a LCS y a solution(), siendo  $O(|obj|) < O(|obj| * |base|)$  por propiedades de O la cota superior de todo el problema es  $O(|base| * |obj|)$

---

<sup>2</sup>Ver 7

<sup>3</sup>Ver 7.1

<sup>4</sup>Ver 7.2

## 4. Ejecución y pruebas

Siendo que el lenguaje utilizado es Python, no es necesaria ninguna compilación. El programa se puede ejecutar directamente mediante el siguiente formato:

```
python tdatp2 <base> <objetivo> <archivo costos>
```

o bien:

```
./tdatp2 <base> <objetivo> <archivo costos>
```

Asimismo se incluye un conjunto de datos de prueba y scripts para ejecutar pruebas de manera sencilla, dichos scripts se encuentran escritos en lenguaje bash para ser ejecutados en entornos UNIX.

Para la primera parte del enunciado, se dan 4 conjuntos de datos, para realizar pruebas con los mismos se debe invocar al script *test.sh* de la siguiente manera:

```
./test.sh [1-4]
```

De similar modo para ejecutar la prueba del ejemplo de alineación de secuencias de ADN, se provee del archivo de datos mencionado con anterioridad y un script de prueba *adntest.sh* cuya invocación es directa:

```
./adntest.sh
```

## 5. Conclusiones

Se diseñó e implementó un algoritmo para analizar las operaciones necesarias para transformar una palabra en otra, considerando una serie de costos asociados.

Se hace un análisis del entorno del problema, para analizar las posibles implicancias de una operación y preveer costos a futuro que puede acarrear efectuar la misma.

## 6. Reformas respecto 1º informe

## 7. Codigos Fuente

### 7.1. LCS.py

```
1 """
2 Initialization costs  $O(\#a+1 * \#b+1)$ 
3 The matrix load costs  $O(\#a * \#b)$ 
4 The while statement costs  $O(\#a * \#y)$ 
5  $O(\#a+1 * \#b+1) = O(\#a*\#b + \#a + \#b + 1)$ 
6  $\#a < \#a*\#b$ 
7  $\#b < \#a*\#b$ 
8  $1 < \#a*\#b$ 
9  $\rightarrow$ 
10  $O(\#a*\#b + \#a + \#b + 1) < O(3*\#a*\#b)$ 
11 Total time compexity using big O properties
12  $O(\#a * \#b)$ 
13 """
14 def lcs(a, b):
15     lengths = [[0 for j in range(len(b)+1)] for i in range(len(a)+1)] # $O(\#a+1 * \#b+1)$ 
16     # row 0 and column 0 are initialized to 0 already
17     # This for statement in total has  $O(\#a * \#b)$ 
18     for i, x in enumerate(a): # $O(\#a)$ 
19         for j, y in enumerate(b): # $O(\#b)$ 
20             if x == y:
21                 lengths[i+1][j+1] = lengths[i][j] + 1 # $O(1)$ 
22             else:
23                 lengths[i+1][j+1] = \
24                     max(lengths[i+1][j], lengths[i][j+1]) # $O(1)$ 
25     # read the substring out from the matrix
26     result = ""
27     x, y = len(a), len(b) # $O(1)$ 
28     """Worst case scenario in each cycle either x or y is modified (not both)
29     Time Compexity:  $O(x*y)$ """
30     while x != 0 and y != 0:
31         if lengths[x][y] == lengths[x-1][y]: # $O(1)$ 
32             x -= 1
33         elif lengths[x][y] == lengths[x][y-1]: # $O(1)$ 
34             y -= 1
35         else:
36             assert a[x-1] == b[y-1] # $O(1)$ 
37             result = a[x-1] + result
38             x -= 1
39             y -= 1
40     return result
```

## 7.2. tdatp2.py

```
1 #!/usr/bin/python
2 # -*- coding: utf-8 -*-
3
4 import sys
5 import LCS
6
7 class Cost(object):
8     """Se utiliza el patron singleton que fue modificado
9     para que no se llame a init en cada llamado a la instancia"""
10
11     _instance = None
12     _init = False
13
14     """O(1)"""
15     def __new__(cls, *args, **kwargs):
16         if not cls._instance:
17             cls._instance = super(Cost, cls).__new__(cls)
18         return cls._instance
19
20     """O(1)"""
21     def __init__(self, file_source = None):
22         if self._init : return #Avoids __init__ being called on every access.
23         self.dict = {}
24         if file_source is not None:
25             self.load_file(file_source)
26         self._init = True
27
28     """O(#Operaciones) en este caso O(6)"""
29     def load_file(self, file_source):
30         #print "el archivo se llama %s \n" % file_source
31         _file = open(file_source)
32         for line in _file:
33             line = line.replace('\n', '').split(':')
34             self.dict[line[0]] = int(line[1])
35
36     """O(1)"""
37     def costo(self, op):
38         return self.dict[op]
39
40     def existe(self, op):
41         return self.dict.has_key(op)
42
43     """TODAS LAS OPERACIONES SON O(1)"""
```

```

44 class Copiar():
45     def __init__(self, char):
46         self.char = char
47
48     def __str__(self):
49         s = "copiar_%s" % self.char
50         return s
51
52 class Reemplazar():
53     def __init__(self, char1, char2):
54         self.char1=char1
55         self.char2=char2
56
57     def __str__(self):
58         s = "reemplazar_%s_%s" %(self.char1, self.char2)
59         return s
60
61 class Borrar():
62     def __init__(self, char):
63         self.char=char
64
65     def __str__(self):
66         s = "borrar_%s" % self.char
67         return s
68
69 class Insertar():
70     def __init__(self, char):
71         self.char=char
72
73     def __str__(self):
74         s = "insertar_%s" % self.char
75         return s
76
77 class Intercambiar():
78     def __init__(self, char1, char2):
79         self.char1=char1
80         self.char2=char2
81
82     def __str__(self):
83         s = "intercambiar_%s_%s" %(self.char1, self.char2)
84
85         return s
86
87 class Terminar():
88     def __str__(self):

```

```

89     return "terminar"
90
91
92 class Problem():
93     """O(1)"""
94     def __init__(self, pal1, pal2):
95         self.base = pal1
96         self.posbase = 0
97         self.objective = pal2
98         self.lcs= LCS.lcs(pal1, pal2) #O(#pal1 * #pal2)
99         self.poslcs = 0
100        self.mem={}
101        self.diff = len(pal2)-len(pal1) #O(1)
102        self.cost=0
103        self.term = False if (self.diff >= 0) else True #O(1)
104        self.inser = self.diff if (self.diff > 0) else 0 #O(1)
105        self.borr = self.diff if (self.diff < 0) else 0 #O(1)
106
107    """O(1)"""
108    def eob(self):
109        return self.posbase == len(self.base)
110
111    def verLcs(self):
112        try:
113            return self.lcs[self.poslcs]
114        except IndexError:
115            return None
116
117    """O(1)"""
118    def verBase(self):
119        return self.base[self.posbase]
120
121    """O(1)"""
122    def verSigBase(self):
123        try:
124            return self.base[self.posbase+1]
125        except IndexError:
126            return None
127
128    """O(1)"""
129    def verObj(self, pos):
130        return self.objective[pos]
131
132    """O(1)"""
133    def verSigObj(self, pos):

```



```

134     try:
135         return self.objective[pos+1]
136     except IndexError:
137         return None
138
139 """O(1)"""
140 def copiar(self):
141     c = Copiar(self.verBase())
142     self.posbase += 1
143     self.cost += Cost().costo('copiar')
144     self.poslcs += 1
145     return [c]
146
147 """O(1)"""
148 def insertar(self, pos):
149     char = self.objective[pos]
150     i = Insertar(char)
151     self.cost += Cost().costo('insertar')
152     _terminar = Cost().existe('terminar')
153     _borrar = Cost().existe('borrar')
154     if self.diff >= 0:
155         if not _borrar and _terminar:
156             self.term = True
157         elif _borrar and _terminar:
158             minim = min(Cost().costo('borrar'), Cost().costo('terminar'))
159             if minim == Cost().costo('terminar'):
160                 self.term = True
161     return [i]
162
163 """O(1)"""
164 def terminar(self):
165     t = Terminar()
166     self.cost += Cost().costo('terminar')
167     return [t]
168
169 """O(1)"""
170 def intercambiar(self):
171     x = self.verBase()
172     y = self.verSigBase()
173     i = Intercambiar(x,y)
174     self.cost += Cost().costo('intercambiar')
175     self.posbase += 2
176     return [i]
177
178 """O(1)"""

```

```

179 def reemplazar(self, pos):
180     r = Reemplazar(self.verBase(), self.verObj(pos))
181     self.cost += Cost().costo('reemplazar')
182     if (self.verBase() == self.verLcs()):
183         self.poslcs += 1
184     self.posbase += 1
185     return [r]
186
187 """O(1)"""
188 def borrar(self):
189     b = Borrar(self.verBase())
190     self.posbase += 1
191     self.cost += Cost().costo('borrar')
192     return [b]
193
194 """O(1)"""
195 def aInsertar(self, pos):
196     """Se analiza el costo de insertar, se supone que la prox seria copia
197     sino no conviene insertar"""
198     costo = Cost().costo('insertar') + Cost().costo('copiar')
199     _terminar = Cost().existe('terminar')
200     _borrar = Cost().existe('borrar')
201     if self.diff <= 0:
202         #Insertar implica que deba o borrar o terminar
203         if not _borrar and _terminar:
204             costo += Cost().costo('terminar') if not self.term else 0
205         elif not _terminar and _borrar:
206             costo += Cost().costo('borrar')
207         else:
208             step = min(Cost().costo('terminar'), Cost().costo('borrar'))
209             if step == Cost().costo('terminar'):
210                 costo += step if not self.term else 0
211             else:
212                 costo += step
213
214     return costo
215
216 """O(1)"""
217 def aBorrar(self, pos):
218     costo = Cost().costo('borrar') + Cost().costo('copiar')
219     if self.diff >= 0:
220         #Borrar implica insertar
221         costo += Cost().costo('insertar')
222     return costo
223

```

```

224 """O(1)"""
225 def aReemplazar(self, pos):
226     costo=Cost().costo('reemplazar')
227
228     if self.verBase()==self.verLcs():
229         costo+=Cost().costo('reemplazar') if Cost().existe('terminar') else 0
230     if self.verSigBase()==self.verSigObj(pos):
231         costo+=Cost().costo('copiar')
232     else:
233         costo+=Cost().costo('reemplazar')
234     return costo
235
236 """O(1)"""
237 def checkintercambio(self, pos):
238     """Evalua si es viable un intercambio"""
239     if not Cost().existe('intercambio'):
240         return False
241     if pos < len(self.objective):
242         """Evaluo caso de intercambio"""
243         b1 = self.verBase()
244         b2= self.verSigBase()
245         o1 = self.objective[pos]
246         o2 = self.verSigObj(pos)
247         if (not b2 is None and not o2 is None) and (b1 == o2) and (b2 == o1):
248             return True
249         return False
250
251 """
252 El peor caso es que me encuentre al principio del string y que no haya ningun
253 elemento cuya copia sea posible por lo que debe recorrer todo el string base
254 O(#base)
255 """
256 def distcopy(self, pos):
257     """Mide la distancia al proximo elemento que se pueda copiar"""
258     dist = 0
259     aux = self.posbase #Guardo la posicion original
260     while not self.eob():
261         if self.verBase()==self.verObj(pos):
262             dist = self.posbase - aux
263             self.posbase += 1
264         self.posbase = aux #Vuelvo a colocarlo en su posicion original
265     return dist
266
267 """
268 Idem anterior, el peor caso es que no haya ningun elemento cuyo intercambio

```

```

269 sea posible.
270 O(#base)
271 """
272 def distinter(self, pos):
273     """Mide la distancia al proximo intercambio(de ser posible)"""
274     dist = 0
275     aux = self.posbase
276     while not self.eob():
277         if self.checkintercambio(pos):
278             dist = self.posbase - aux
279             self.posbase +=1
280             self.posbase = aux
281         return dist
282
283 """
284 Mide el minimo en un conjunto de operaciones
285 O(#l)
286 En esta implementacion podemos decir que es:
287 O(2)=O(1)
288 """
289 def min(self, l):
290     minimo = None
291     for i in l:
292         if (i[0]==0):
293             continue
294         if (minimo is None) or (i[0]*Cost().costo(i[1]) < Cost().costo(minimo)):
295             minimo = i[1]
296     return minimo
297
298 """
299 O(#l)
300 A efectos de esta implementacion:
301 O(3)=O(1)
302 """
303 def min2(self, l):
304     minimo = None
305     costo = None
306     for i in l:
307         if i[0] is None:
308             continue
309         if (minimo is None) or (i[0] < costo):
310             minimo = i[1]
311             costo = i[0]
312     return minimo
313

```

```

314 """
315 Determina cual es la mejor manera de obtener el caracter actual
316 con los elementos disponibles en la palabra base
317 O(1)
318 """
319 def solve(self, pos, aux=None):
320     r = []
321     if self.eob(): #O(1)
322         """No hay otra opcion mas que insertar, ya que se agotaron
323         los caracteres disponibles en la base"""
324         if Cost().existe('insertar'):
325             return self.insertar(pos) #O(1)
326
327     #O(1)
328     if (self.verBase() == self.objective[pos]) \
329         and Cost().existe('copiar'):
330         """Es el caso de copiar"""
331         aux = self.checkintercambio(pos) #O(1)
332         """Evaluo si en vez de copiar se puede intercambiar y si es mas optimo"""
333         #O(1)
334         if not aux or (aux and \
335             self.min([(1, 'copiar'), (1, 'intercambiar')]) == 'copiar'):
336             """Evaluo si en vez de copiar se puede reemplazar"""
337             #O(1)
338             if (not Cost().existe('reemplazar')) or \
339                 (self.min([(1, 'copiar'), (1, 'reemplazar')]) == 'copiar'):
340                 return self.copiar() #O(1)
341             else:
342                 """No tiene sentido que reemplazar sea mas eficiente,
343                 igual se implementa"""
344                 return self.reemplazar(pos) #O(1)
345
346     #O(1)
347     if pos < len(self.objective):
348         """Evaluo caso de intercambio"""
349         if self.checkintercambio(pos):
350             r = self.intercambiar() #O(1)
351             """
352             Se guarda en dos posiciones de memoria dado que al intercambiar
353             estamos solucionando dos posiciones del objetivo
354             """
355             try:
356                 self.mem[pos+1] = self.mem[pos-1] + r #O(1)
357             except KeyError:
358                 if (aux is not None):

```

```

359         self.mem[pos+1]=aux+r          #O(1)
360     else:
361         pass
362     return r
363
364     if Cost().existe('copiar'):
365         costoInsertar = None
366         costoBorrar = None
367         costoReemplazar = None
368         if self.verBase() == self.verLcs(): #O(1)
369             #Evaluó insertar, y la siguiente operacion seria una copia
370             costoInsertar = self.aInsertar(pos) #O(1)
371         if self.verSigBase() == self.verLcs() :
372             #Evaluó borrar, y la siguiente operacion seria una copia
373             costoBorrar = self.aBorrar(pos)
374             costoReemplazar = self.aReemplazar(pos)
375         #O(1)
376         op = self.min2([(costoReemplazar, 'r'),(costoInsertar, 'i'),(costoBorrar, 'b')])
377         if op == 'r':
378             return self.reemplazar(pos)
379         elif op == 'b':
380             r += self.borrar() #O(1)
381             r += self.solve(pos, r) #O(1)
382             return r
383         else:
384             return self.insertar(pos) #O(1)
385
386     print "No debiera llegar a este punto—"
387
388     """
389     Se emplea programacion dinamica usando memorizacion.
390     Se hacen tantas llamadas a la funcion como caracteres a obtener de alli
391     O(#Objetivo)
392     """
393     def solution(self, pos):
394         """Como premisa supongo que ya poseo como conseguir una solucion anterior
395         """
396         if pos == 0:
397             return self.solve(pos)
398
399         if not self.mem.has_key(pos-1):
400             self.mem[pos-1] = self.solution(pos-1)
401         if not self.mem.has_key(pos):
402             self.mem[pos] = self.mem[pos-1] + self.solve(pos)
403         if pos == len(self.objective)-1 and not self.eob():

```

```

404         """Ya obtuvimos la solucion y aun hay elementos en la base que deben ser
405         descartados"""
406         self.mem[pos] += self.terminar()
407         return self.mem[pos]
408
409 """
410 Siendo #pal2 < #pal1 * #pal2
411  $O(\#pal1 * \#pal2) + O(\#pal2) < 2 O(\#pal1 * \#pal2)$ 
412  $O(\#pal1 * \#pal2)$ 
413 """
414 def main():
415     print "Teoria y Algoritmos 1-[75.29]"
416     print "TP2-Distancia de Edicion"
417     print "Autores: Alejandro Pernin(92216) - Lautaro Medrano(90009)\n"
418     s1 = Cost(file_source=sys.argv[3])
419     pal = sys.argv[1]
420     pal2 = sys.argv[2]
421     p = Problem(pal, pal2)  $O(\#pal1 * \#pal2)$ 
422     s = p.solution(len(pal2)-1)  $O(\#pal2)$ 
423     for i in enumerate(s):
424         print i[0]+1, ')', i[1]
425         print "\nEl costo es: %s" % str(p.cost)
426 if __name__ == '__main__':
427     main()

```