Nombre: Juan Manuel Castillo Nievas

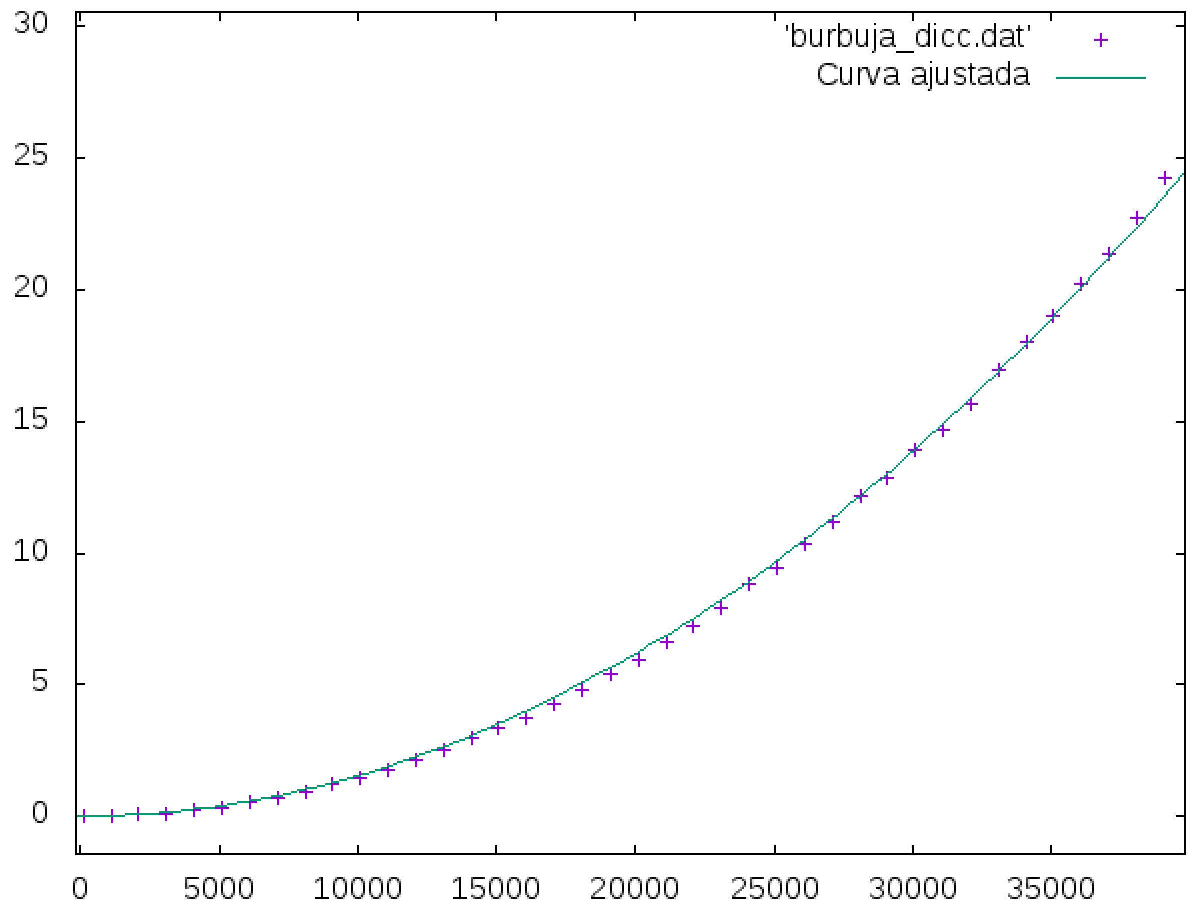
6.1 Algoritmo de ordenación por burbuja

Al ordenar el diccionario de palabras empezando por un subconjunto de 100 palabras y acabando en un subconjunto de 50000 palabras, sumando en cada iteración 1000 palabras he obtenido los siguientes tiempos:

100 0.000195017
1100 0.0181199
2100 0.0551645
3100 0.124558
4100 0.224634
5100 0.353199
6100 0.512754
7100 0.707994
8100 0.933186
9100 1.21558
10100 1.47704
11100 1.77807
12100 2.11585
13100 2.51198
14100 2.98398
15100 3.33023
16100 3.76813
17100 4.30374
18100 4.82441
19100 5.39584
20100 5.91716
21100 6.61692
22100 7.2515
23100 7.92353
24100 8.80204
25100 9.47576
26100 10.3592
27100 11.1818
28100 12.1782
29100 12.8679
30100 13.9468
31100 14.691
32100 15.6805
33100 16.9895
34100 18.0019
35100 19.0226
36100 20.1964
37100 21.3495
38100 22.7066
39100 24.2329
40100 27.3501
41100 26.0613
42100 27.3384

43100 28.6839
44100 30.5163
45100 32.3976
46100 34.3551
47100 35.0441
48100 39.8167
49100 39.2711

Realizando un análisis híbrido he obtenido una función cuadrática f(x) = a*x*x



en la cual el parámetro a me ha dado el valor de $1.54192 * e^{-8}$

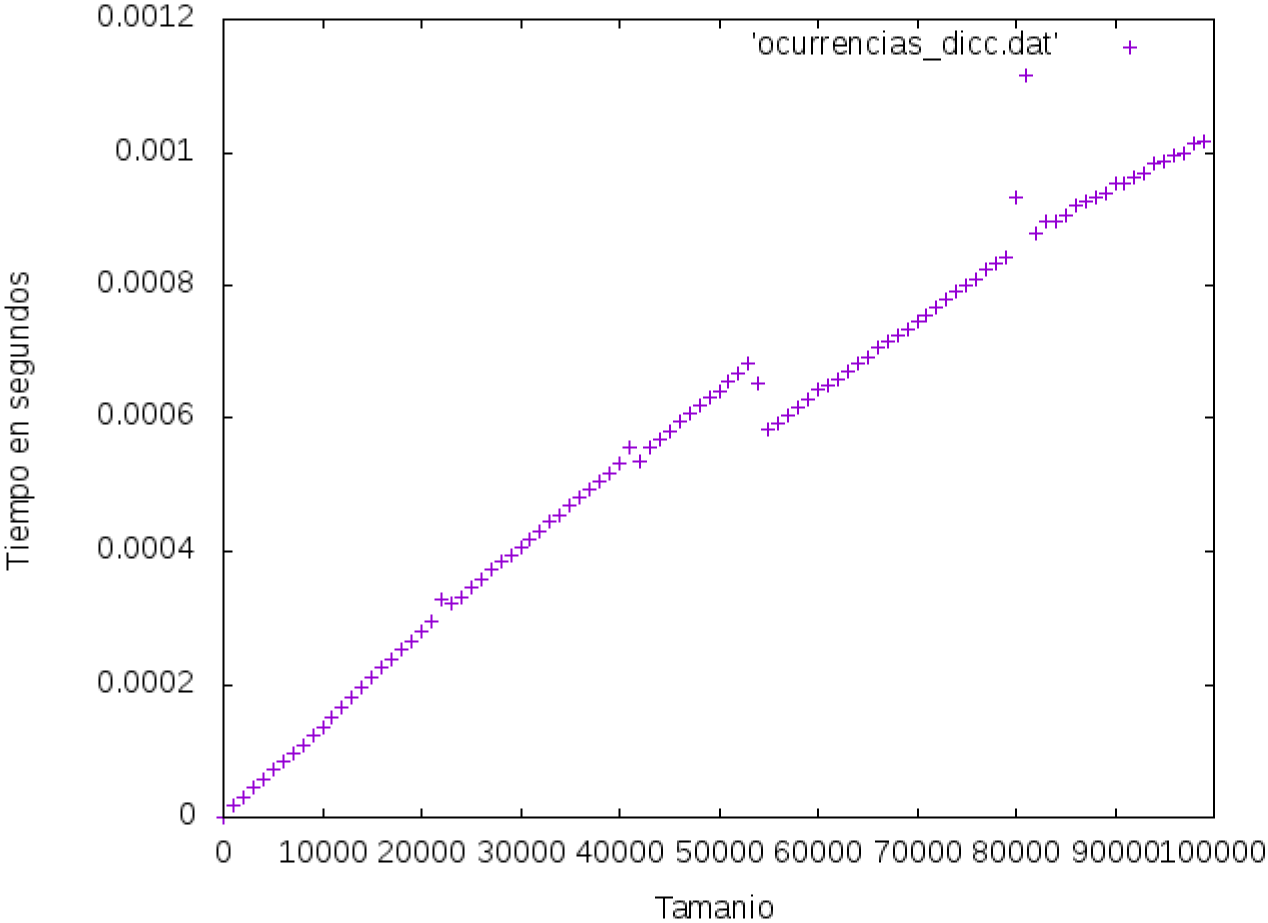
6.2.1 Análisis de eficiencia ocurrencias.cpp

<u>En el caso de “lema.txt” he obtenido los siguientes resultados y la siguiente gráfica:</u>

10 1.39e-06
1010 1.9483e-05
2010 3.1309e-05
3010 4.379e-05
4010 5.8014e-05
5010 7.1097e-05
6010 8.3115e-05
7010 9.5736e-05
8010 0.000109064
9010 0.000123066

10010 0.000135809
11010 0.000151521
12010 0.000165797
13010 0.000180389
14010 0.000195762
15010 0.000210214
16010 0.000224402
17010 0.000238226
18010 0.000251665
19010 0.000265981
20010 0.000280549
21010 0.000294739
22010 0.000327652
23010 0.000320646
24010 0.000332207
25010 0.000345555
26010 0.000359046
27010 0.000373117
28010 0.000384609
29010 0.000394564
30010 0.000405678
31010 0.000418008
32010 0.000430145
33010 0.000446488
34010 0.000454949
35010 0.000467671
36010 0.00047974
37010 0.000493479
38010 0.00050591
39010 0.000518542
40010 0.000531698
41010 0.000555189
42010 0.000536662
43010 0.00055515
44010 0.000568311
45010 0.000580254
46010 0.000594946
47010 0.000608182
48010 0.000619881
49010 0.00063033
50010 0.000640523
51010 0.000655019
52010 0.000668249
53010 0.00068266
54010 0.000651374
55010 0.000582539
56010 0.00059339
57010 0.000605475
58010 0.000617082
59010 0.000628173
60010 0.000643028
61010 0.000649624

62010 0.000658695
63010 0.000670405
64010 0.000682108
65010 0.000692623
66010 0.000707691
67010 0.000714668
68010 0.000723368
69010 0.000734323
70010 0.000745673
71010 0.000754933
72010 0.000768172
73010 0.00077879
74010 0.000790287
75010 0.000800718
76010 0.000809948
77010 0.000823255
78010 0.000832038
79010 0.00084243
80010 0.000932957
81010 0.00111709
82010 0.000879394
83010 0.000896823
84010 0.000895264
85010 0.000906062
86010 0.000920686
87010 0.000927397
88010 0.000933379
89010 0.000938993
90010 0.000953383
91010 0.000954118
92010 0.000961694
93010 0.000969589
94010 0.000982593
95010 0.000986835
96010 0.000996352
97010 0.000999846
98010 0.00101317
99010 0.00101552

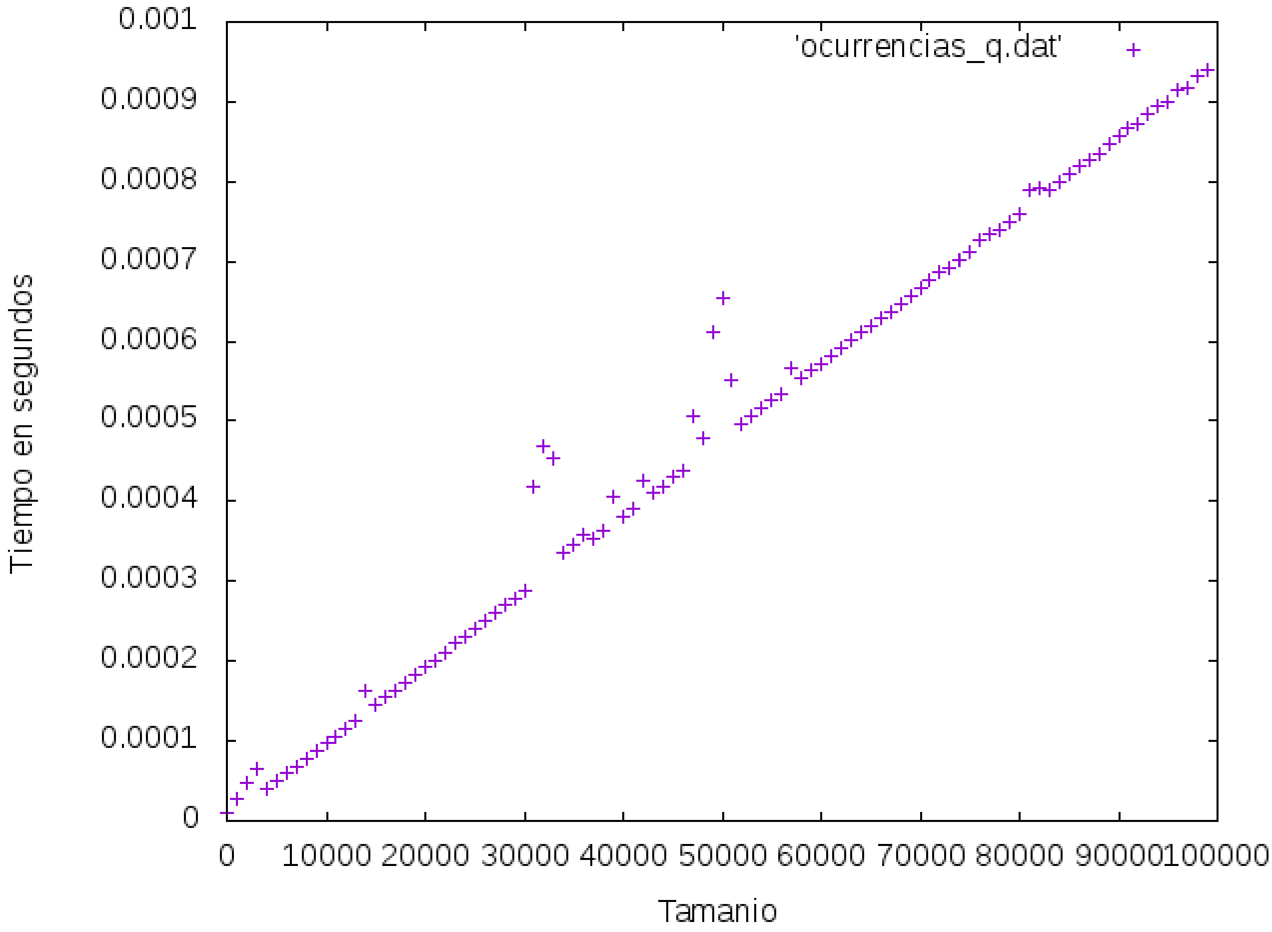


En el caso de "quijote.txt" he obtenido lo siguiente:

10 1.0847e-05
1010 2.8785e-05
2010 4.6438e-05
3010 6.5248e-05
4010 4.1292e-05
5010 4.9963e-05
6010 5.9244e-05
7010 6.7644e-05
8010 7.7291e-05
9010 8.7108e-05
10010 9.6731e-05
11010 0.000106076
12010 0.000115792
13010 0.000125726
14010 0.000161784
15010 0.000144859
16010 0.000154307
17010 0.000164001
18010 0.000173595
19010 0.000182838
20010 0.000192045
21010 0.000201146

22010 0.00021163
23010 0.000221894
24010 0.000230953
25010 0.000240993
26010 0.000250693
27010 0.000260156
28010 0.000269473
29010 0.000278638
30010 0.000288135
31010 0.000419277
32010 0.00046785
33010 0.000454272
34010 0.000335171
35010 0.000344956
36010 0.000358787
37010 0.00035426
38010 0.000363981
39010 0.000406937
40010 0.000381798
41010 0.000391997
42010 0.000425333
43010 0.000410728
44010 0.000419539
45010 0.000430128
46010 0.000438569
47010 0.000505999
48010 0.000478622
49010 0.000611678
50010 0.00065537
51010 0.000551516
52010 0.000497231
53010 0.000507026
54010 0.000515836
55010 0.000525899
56010 0.000533341
57010 0.000566189
58010 0.000553265
59010 0.000562812
60010 0.000571917
61010 0.000581651
62010 0.000590741
63010 0.000601048
64010 0.000612728
65010 0.000619651
66010 0.000629142
67010 0.000636369
68010 0.000647705
69010 0.000657079
70010 0.000667473
71010 0.000676422
72010 0.000685561
73010 0.000692815

74010 0.000702136
75010 0.000712114
76010 0.000726218
77010 0.000733084
78010 0.000739758
79010 0.000749278
80010 0.000758237
81010 0.000789649
82010 0.000791147
83010 0.000789996
84010 0.00079933
85010 0.000809555
86010 0.000818499
87010 0.000828135
88010 0.000834226
89010 0.00084693
90010 0.000856011
91010 0.000867578
92010 0.000870998
93010 0.00088442
94010 0.000894247
95010 0.000899659
96010 0.000914833
97010 0.000918175
98010 0.000931142
99010 0.000939885



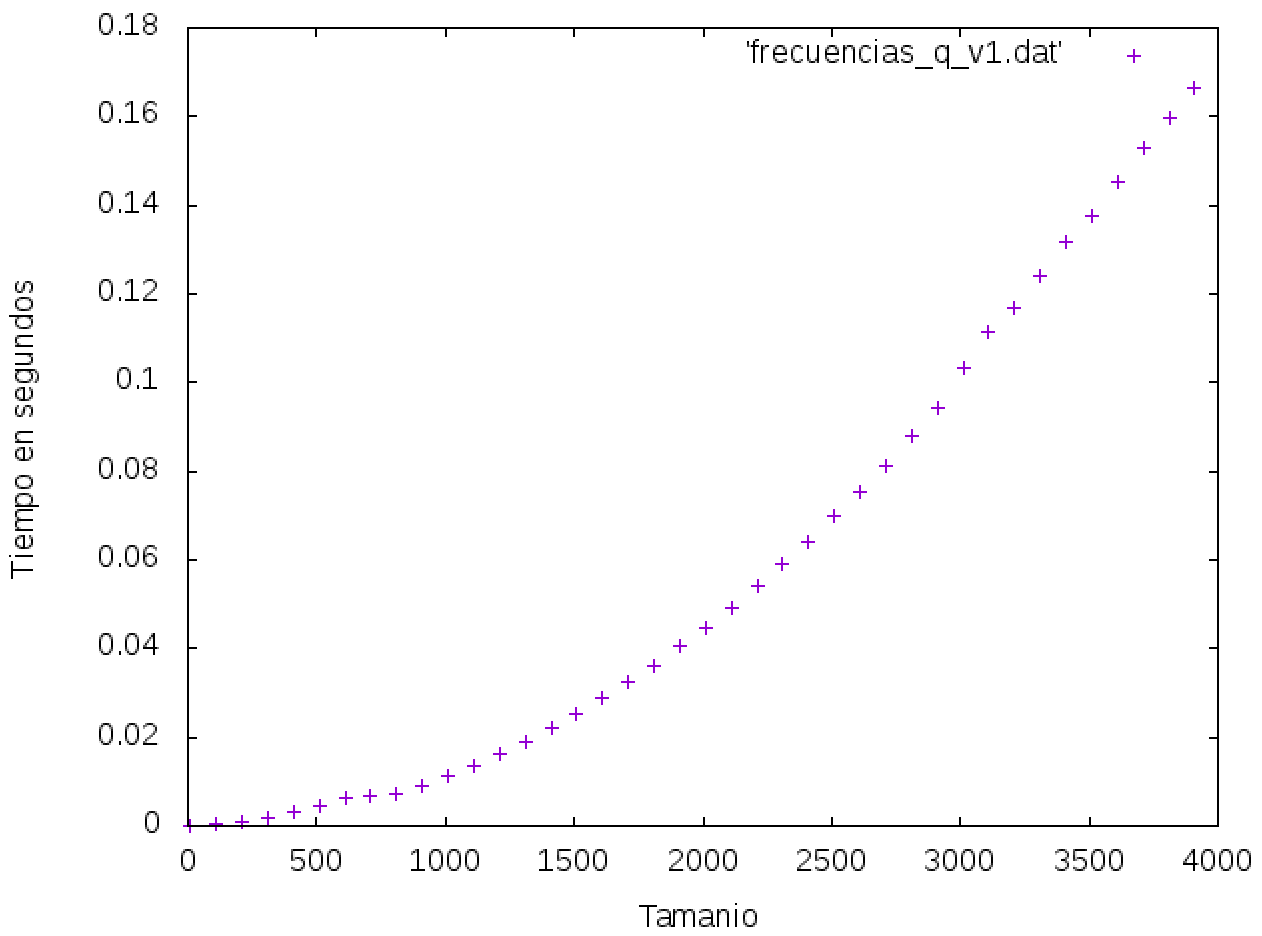
En ambos ficheros he obtenido una gráfica casi lineal, salvo algunos casos excepcionales en los que el tiempo varía de manera incoherente. Además no apreciamos diferencias entre los tiempos muy grandes de un fichero a otro.

6.2.2 Análisis de eficiencia frecuencias.cpp

En la versión 1 he obtenido los siguientes resultados:

10 1.632e-05
110 0.000239781
210 0.000772884
310 0.00167016
410 0.00303451
510 0.00445307
610 0.00619368
710 0.00688629
810 0.00729747
910 0.00915683
1010 0.0112528
1110 0.0136537
1210 0.0162674
1310 0.0189534
1410 0.0220575
1510 0.0254338
1610 0.0286914
1710 0.0324266
1810 0.0362889
1910 0.0404387
2010 0.0446929
2110 0.0493365
2210 0.053912
2310 0.0590597
2410 0.0640897
2510 0.0697276
2610 0.0754381
2710 0.0811608
2810 0.0879666
2910 0.0944309
3010 0.103184
3110 0.111216
3210 0.116721
3310 0.1239
3410 0.131555
3510 0.137476
3610 0.145235
3710 0.153079
3810 0.159588

3910 0.166547



<u>Teóricamente:</u>

```
void contar_frecuencias_V1( vector<string> & libro, int ini, int fin,
                        vector<string> &pal, vector<int> & frec ){
  int cuantas;
  for (int i = ini; i<fin; i++){

     cuantas = contar_hasta(libro,ini,fin,libro[i]);
     pal.push_back(libro[i]);
     frec.push_back(cuantas);
     }
}

int contar_hasta( vector<string> & V, int ini, int fin, string & s) {
  int cuantos = 0;
  for (int i=ini; i< fin ; i++)
    if (V[i]==s) {
     cuantos ++;
    }
  return cuantos;
}
```

Empezamos por la parte más interna, en este caso analizamos primero la función contar_hasta:
$T_{it} = T_{COMP} + T_{CUERPO} + T_{INC}$

- La comparación es de O(1)
- El cuerpo lo forma un *if* que solo tiene una comparación y una asignación, también de O(1)
- La incrementación también es de O(1)

$T_{bi} = \sum_{i=0}^{fin-1} O(1)$ = 1 + 1 + 1 … + 1 = fin que pertenece a 0(n)

Ahora analizamos la función contar_frecuencias_v1:
$T_{it} = T_{COMP} + T_{CUERPO} + T_{INC}$

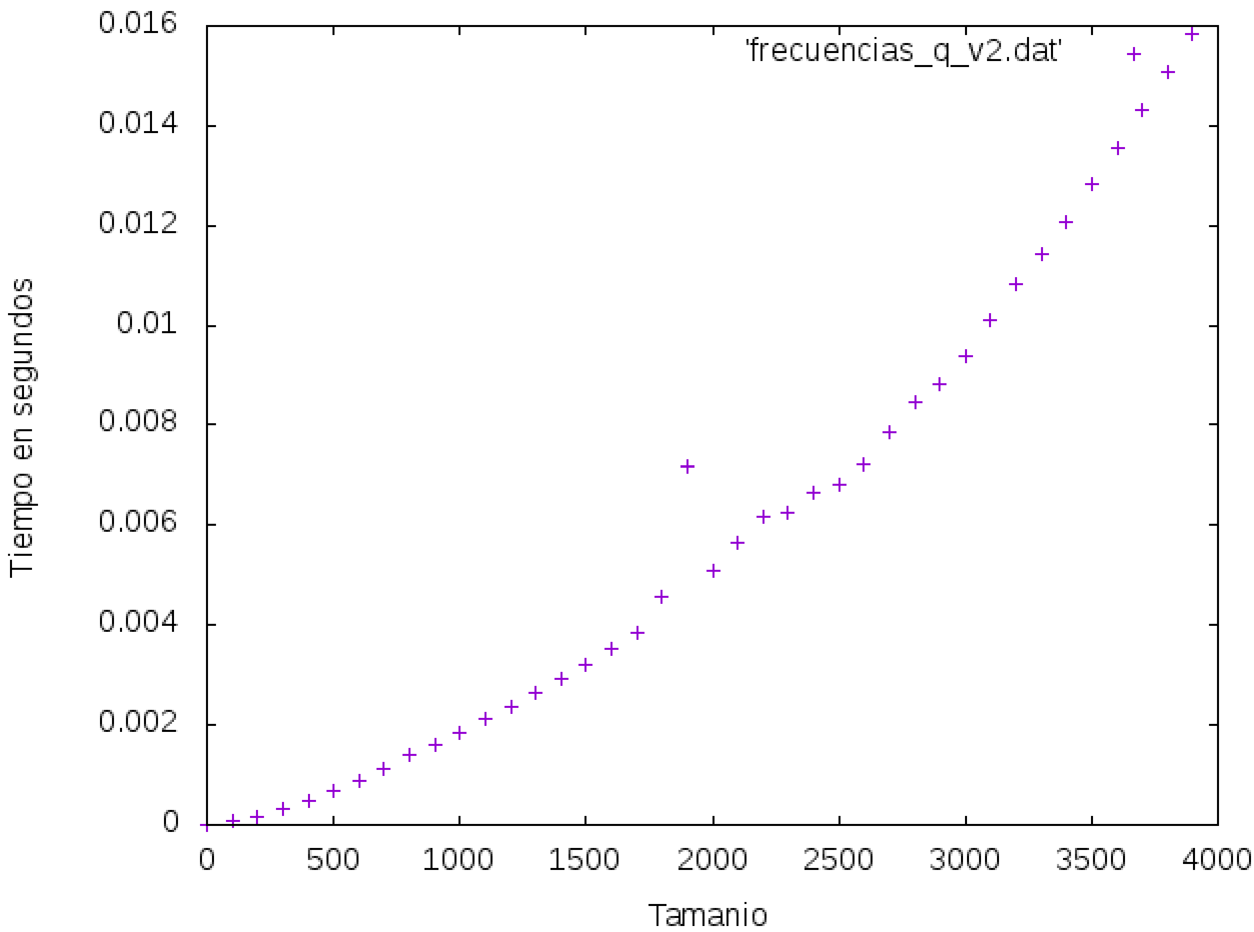
- La comparación es de O(1)
- El cuerpo lo forman 3 sentencias:
  - La función contar_hasta es de O(n)
  - Las otras dos sentencias son de O(1)
- La incrementación es de O(1)

$T_{bi} = \sum_{i=0}^{fin-1} O(n)$ = n repetida *fin* veces que pertenece a $0(n^2)$

En el caso de la versión 2:

0 1.92e-07
100 7.4322e-05
200 0.000166148
300 0.000337493
400 0.000468127
500 0.000677534
600 0.000884485
700 0.00111085
800 0.00138739
900 0.00158901
1000 0.00185268
1100 0.00210946
1200 0.00236097
1300 0.00263921
1400 0.00293707
1500 0.0032084
1600 0.00352338
1700 0.00385622
1800 0.00456785
1900 0.00717785
2000 0.00508738
2100 0.00566544
2200 0.00618707
2300 0.00623639
2400 0.00666906
2500 0.00682273
2600 0.00721435
2700 0.00787126
2800 0.00845198
2900 0.00880426
3000 0.00937868
3100 0.0101212
3200 0.0108287
3300 0.0114148
3400 0.0120758

3500 0.012827
3600 0.0135346
3700 0.014321
3800 0.0150842
3900 0.0158499



Teóricamente:

```
void contar_frecuencias_V2( vector<string> & libro, int ini, int fin, vector<string> &pal, vector<int> & frec ){
  int pos;
  for (int i = ini; i<fin; i++){

     pos = buscar(pal, libro[i]);
     if (pos==POS_NULA) {
        pal.push_back(libro[i]);    // Analisis amortizado O(1)
        frec.push_back(1);    // Analisis amortizado O(1)
        }
     else {
        frec[pos]++;
        }

     }
}
```

Empezamos analizando la parte más interna.

- La función buscar es de O(n) como bien indica el archivo.
- El bloque if/else tiene sentencias simples de comparación y asignación que son todas de O(1)

Entonces analizando el bucle principal:

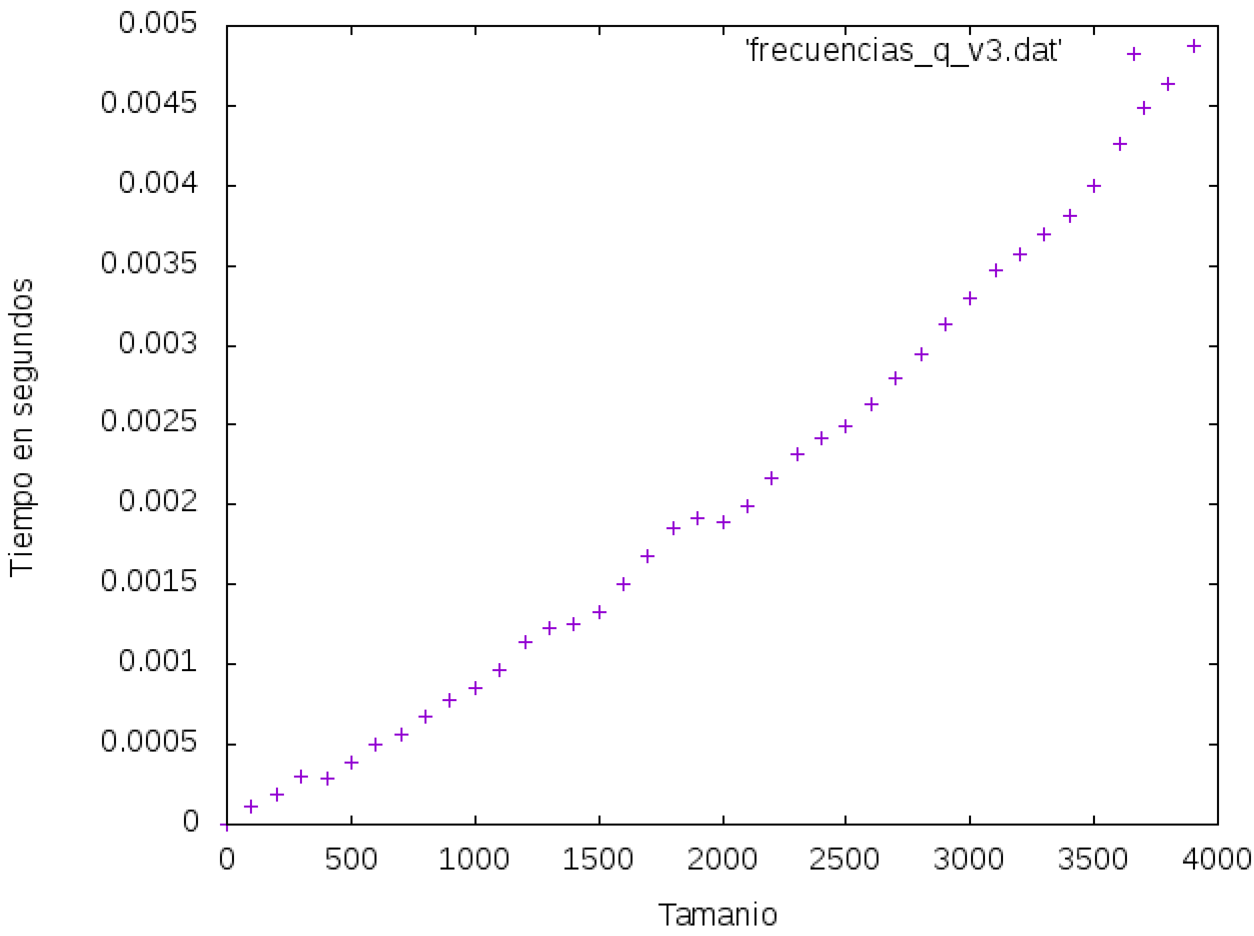
$T_{it} = T_{COMP} + T_{CUERPO} + T_{INC}$

- La comparación es de O(1)
- El cuerpo es de O(n) por la función buscar que es el orden máximo
- La incrementación es de O(1)

$T_{bi} = \sum_{i=0}^{fin-1} O(n)$ = n repetida *fin* veces que pertenece a $0(n^2)$

<u>En el caso de la versión 3:</u>

0 1.86e-07
100 0.000109373
200 0.000193459
300 0.000306128
400 0.000287546
500 0.000384287
600 0.000502221
700 0.000570127
800 0.000677194
900 0.000776681
1000 0.000848428
1100 0.000963266
1200 0.00113667
1300 0.00123343
1400 0.00124839
1500 0.00132559
1600 0.00150114
1700 0.00168182
1800 0.00185252
1900 0.0019167
2000 0.00188844
2100 0.00199363
2200 0.00216454
2300 0.00231217
2400 0.00242273
2500 0.00248786
2600 0.00262963
2700 0.00278951
2800 0.00294006
2900 0.00313653
3000 0.00330014
3100 0.00346972
3200 0.00357312
3300 0.00369563
3400 0.00380877
3500 0.00399438
3600 0.00426039
3700 0.00448786
3800 0.00463974
3900 0.00487548



__Teóricamente:__

```
void contar_frecuencias_V3( vector<string> & libro, int ini, int fin,
                        vector<string> &pal, vector<int> & frec ){
  vector<string>::iterator pos;
  for (int i = ini; i<fin; i++){

     pos = lower_bound(pal.begin(), pal.end(), libro[i]);  // O(log (n) ), n tama del vector  Busqueda Binaria
     if ((pos==pal.end()) || (*pos!=libro[i])) {
         frec.insert(frec.begin() + (pos-pal.begin()), 1);   //O(n)
         pal.insert(pos, libro[i]);           //O (n)
         }
     else {
         frec[pos-pal.begin()]++; // O(1)
         }

     }
}
```

Analizamos el bucle for principal directamente, pues la parte más interna ya viene analizada.

$T_{it} = T_{COMP} + T_{CUERPO} + T_{INC}$

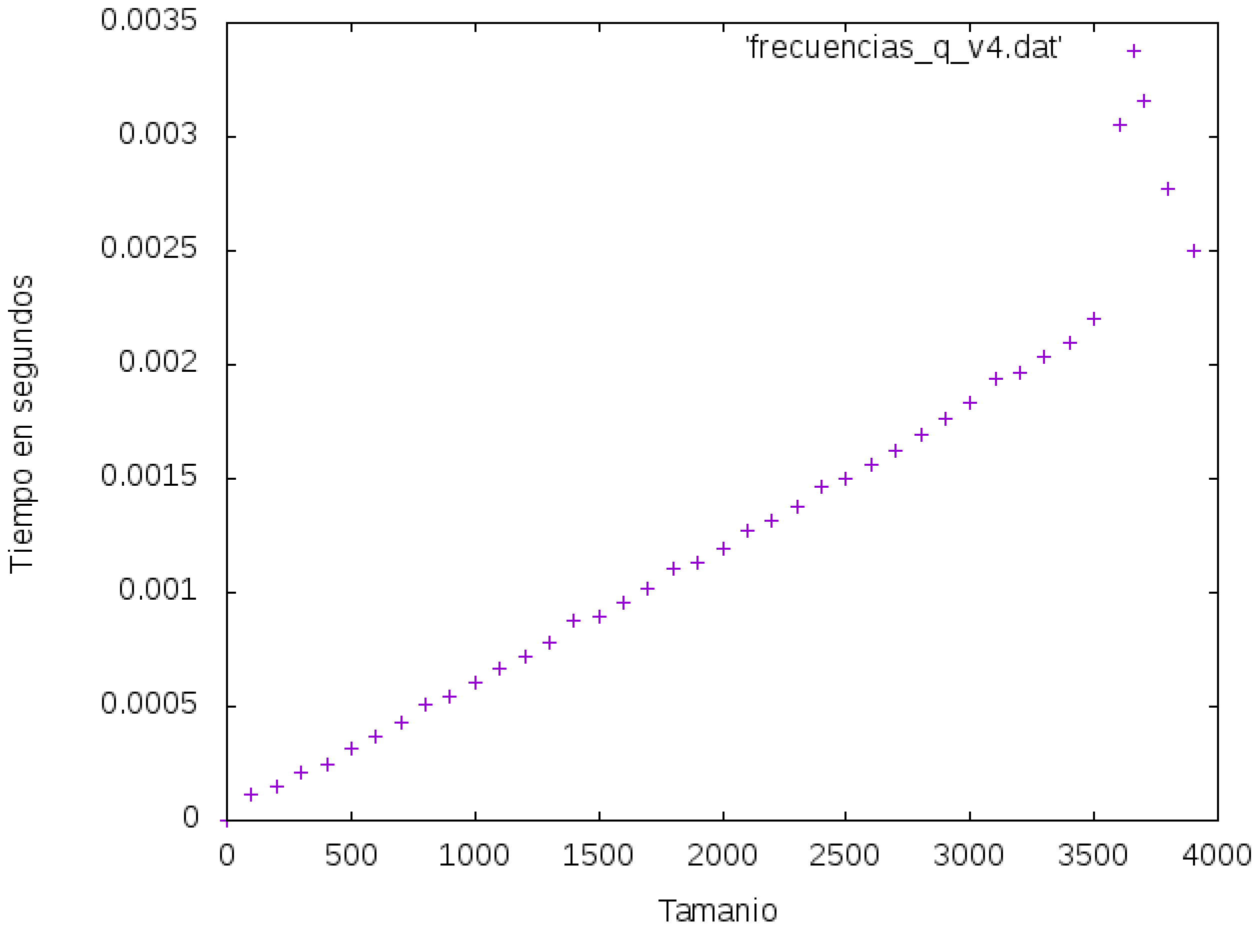
- La comparación es de O(1)
- El cuerpo es de O(n) porque es el orden máximo que encontramos (frec.insert(frec.begin() + (pos-pal.begin()), 1); y pal.insert(pos, libro[i]);)
- La incrementación es de O(1)

$T_{bi} = \sum_{i=0}^{fin-1} O(n)$ = n repetida *fin* veces que pertenece a $0(n^2)$

En el caso de la versión 4:

0 1.362e-06
100 0.000109984
200 0.00015306
300 0.000209277
400 0.000248019
500 0.000316889
600 0.000369629
700 0.000428986
800 0.000512918
900 0.000545654
1000 0.000606636
1100 0.000666327
1200 0.000720873
1300 0.000781575
1400 0.000873988
1500 0.000894514
1600 0.000959219
1700 0.00101523
1800 0.00110172
1900 0.00113535
2000 0.00119357
2100 0.00127488
2200 0.00131343
2300 0.001375
2400 0.00146073
2500 0.00149734
2600 0.00155914
2700 0.00162279
2800 0.00169075
2900 0.00176462
3000 0.00183205
3100 0.00193497
3200 0.00196892
3300 0.00203392
3400 0.00209681
3500 0.00219952
3600 0.00304845
3700 0.0031554
3800 0.00276911

3900 0.0025024



<u>Teóricamente:</u>

```
void contar_frecuencias_V4( vector<string> & libro, int ini, int fin,
                         vector<string> &pal, vector<int> & frec ){

  map<string,int> M;
  for (int i = ini; i<fin; i++)
     M[libro[i]]++;        // O( log(n) )

  map<string,int>::iterator it;
  for (it = M.begin(); it!= M.end(); ++ it){ // Bucle O(k log k) siendo k el numero de palabras distintas
    pal.push_back( (*it).first );
    frec.push_back( (*it).second );
     }
}
```

En esta versión ya aparecen los dos bucles analizados con lo cual el orden de la función es el máximo de los dos, siendo este el O(n log (n)).

Conclusión:
Se puede apreciar que en la versión 1 el algoritmo tiene orden cuadrático, y a medida que vamos aumentando las versiones los algoritmos van haciéndose más lineales, siendo la versión 2 cuadrática, la versión 3 cuadŕatica pero más apuntando a lo lineal (al menos en la práctica) y la versión 4 siendo lineal (según la práctica, en la teoría sería de orden n*log(n)). Además, la versión 4 tarda menos que la versión 3, y esta tarda menos que la versión 2. Así mismo, la versión 1 es la que más segundos tarda y la versión 4 la que menos tarda.

6.2.3 Análisis de eficiencia teórico y práctico de los algoritmos de ordenación burbuja, inserción y selección

- Burbuja:
  - Teóricamente:

```
 int i, j;
 string aux;
 for (i = inicial; i < final - 1; i++){
   for (j = final - 1; j > i; j--)
       if (T[j] < T[j-1])
         {
           aux = T[j];
           T[j] = T[j-1];
           T[j-1] = aux;
         }
}
```

$T_{itj} = T_{comp} + T_{cuerpo} + T_{inc}$ = O(1) + O(1) + O(1) = (max(1,1,1)) = O(1)

$T_{bj} = \sum_{final-1}^{0} O(1)$ = 1 + 1 + 1 … + 1 = n que pertenece a O(n)

$T_{iti} = T_{comp} + T_{cuerpo} + T_{inc}$ = O(1) + O(n) + O(1) = (max(1,n,1)) = O(n)

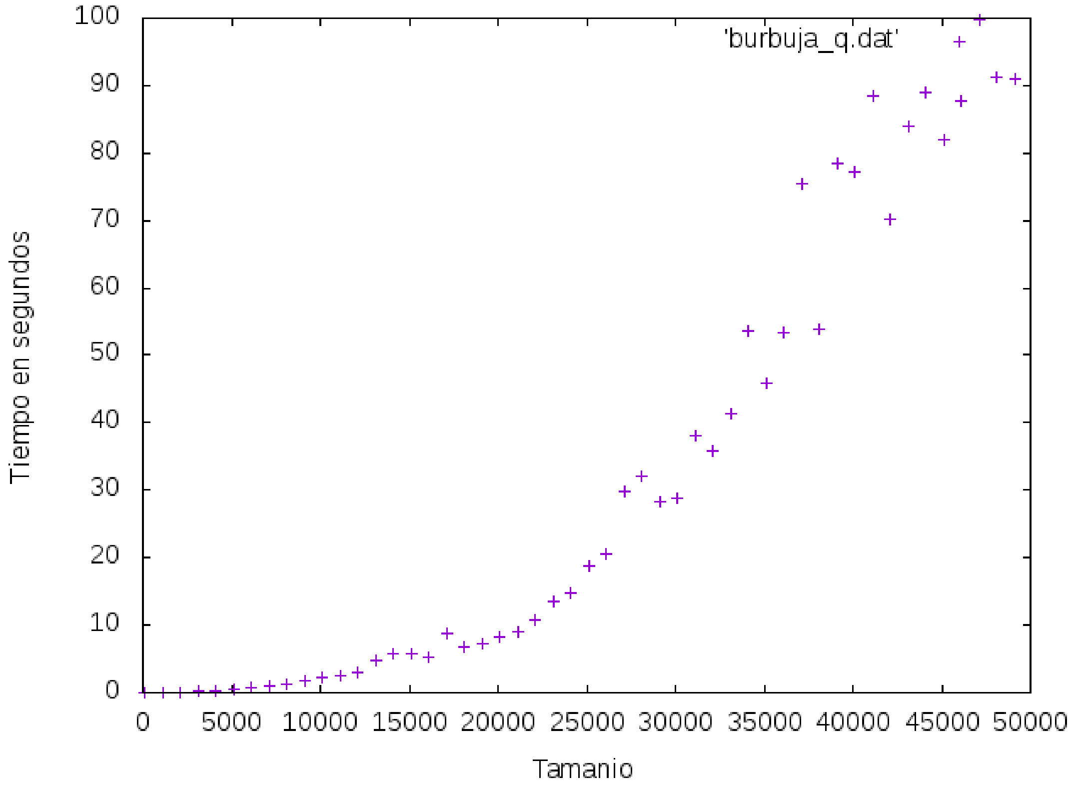
$T_{bi} = \sum_{inicial=0}^{final-1} O(n)$ = n + n + n … + n = n^2 que pertenece a $O(n^2)$

  - Prácticamente:

En el primer ejercicio hemos visto que ordenando el diccionario de palabras aumentando el número de subconjuntos de palabras cada vez obteníamos un orden cuadrático que coincide con el teórico. Si ejecutamos el algoritmo con “quijote.txt” obtenemos algo similar:

100 0.000249171
1100 0.0274445
2100 0.0906782
3100 0.188482
4100 0.326274
5100 0.506442
6100 0.727616
7100 1.00517
8100 1.32133
9100 1.7554
10100 2.17636
11100 2.45516
12100 2.98393
13100 4.6796
14100 5.78295
15100 5.65703
16100 5.33481
17100 8.83645

18100 6.69357
19100 7.31045
20100 8.19357
21100 8.98924
22100 10.6573
23100 13.5772
24100 14.8925
25100 18.8439
26100 20.4496
27100 29.9382
28100 31.9888
29100 28.2721
30100 28.8684
31100 38.0768
32100 35.8252
33100 41.3966
34100 53.5274
35100 45.9795
36100 53.3392
37100 75.4167
38100 53.8997
39100 78.503
40100 77.2628
41100 88.4193
42100 70.0849
43100 83.9829
44100 88.9402
45100 81.9255
46100 87.7409
47100 99.6839
48100 91.2608
49100 91.0596
Podemos observar que al ordenar el Quijote que está más desordenado la gráfica los tiempos son mucho mayores y no llega a ser muy cuadrática, los tiempos están más alejados unos de otros.



- <u>Selección:</u>
  - <u>Teóricamente:</u>

```
for (i=ini; i<n-1; i++)
{
 min=i;
 for(j=i+1; j<n; j++)
    if(T[min] > T[j])
          min=j;

  aux=T[min];
  T[min]=T[i];
  T[i]=aux ;
}
```

$T_{itj} = T_{comp} + T_{cuerpo} + T_{inc} = O(1) + O(1) + O(1) = (\max(1,1,1)) = O(1)$

$T_{bj} = \sum_{i+1}^{n-1} O(1) = 1 + 1 + 1 \ldots + 1 = n – 2 - i$ que pertenece a $O(n - i)$

$T_{iti} = T_{comp} + T_{cuerpo} + T_{inc} = O(1) + O(n - i) + O(1) = (\max(1,n,1)) = O(n - i)$

$T_{bi} = \sum_{ini=0}^{n-2} O(n-i) = n + (n-1) + (n-2) \ldots + 3 + 2= (n-2)(n-1)/2$ que pertenece a $O(n^2)$

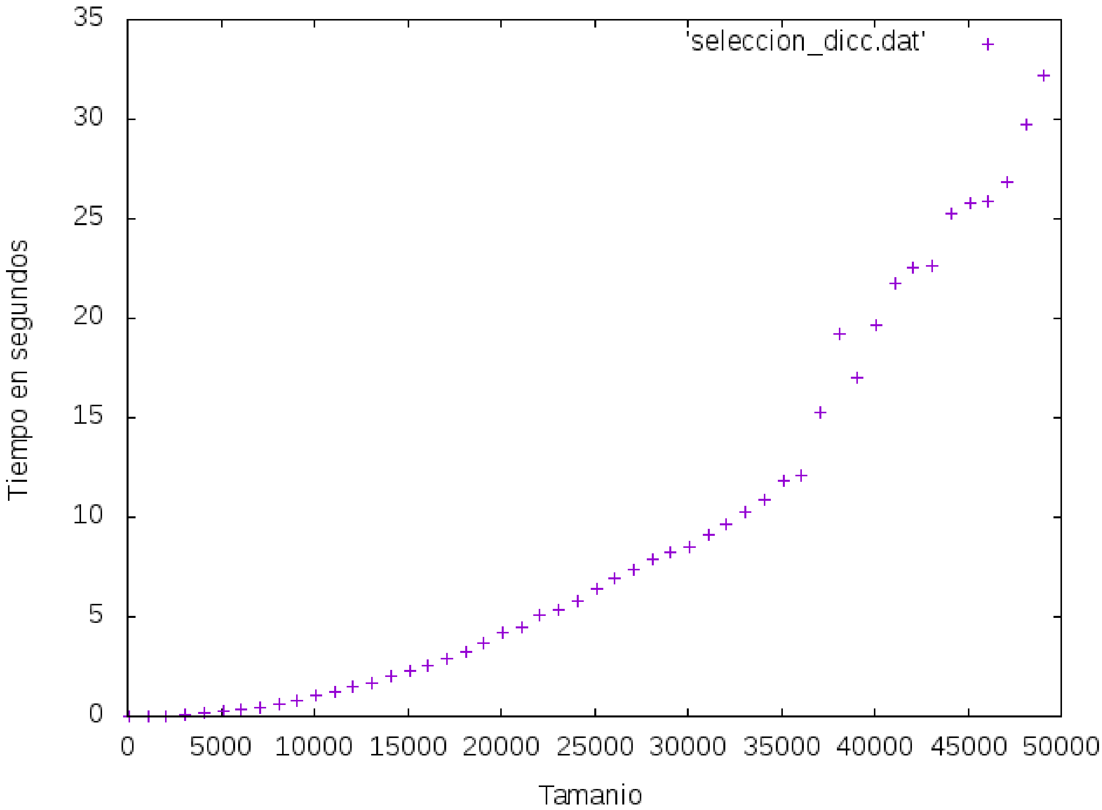
  - <u>Prácticamente:</u>

Al ordenar por selección el diccionario nos salen los siguientes datos y la siguiente gráfica:

100 0.000101834
1100 0.0116923
2100 0.0434355

3100 0.0941512
4100 0.164722
5100 0.253028
6100 0.364445
7100 0.480788
8100 0.62375
9100 0.80298
10100 1.01783
11100 1.20821
12100 1.47649
13100 1.70867
14100 2.00727
15100 2.28494
16100 2.57621
17100 2.90499
18100 3.2338
19100 3.64332
20100 4.20491
21100 4.49391
22100 5.10287
23100 5.34031
24100 5.76273
25100 6.38516
26100 6.90501
27100 7.397
28100 7.88573
29100 8.26683
30100 8.50649
31100 9.14728
32100 9.60902
33100 10.3048
34100 10.8583
35100 11.82
36100 12.1137
37100 15.2881
38100 19.2442
39100 17.0459
40100 19.6133
41100 21.7251
42100 22.5591
43100 22.615
44100 25.284
45100 25.8322
46100 25.8806
47100 26.8049
48100 29.7413

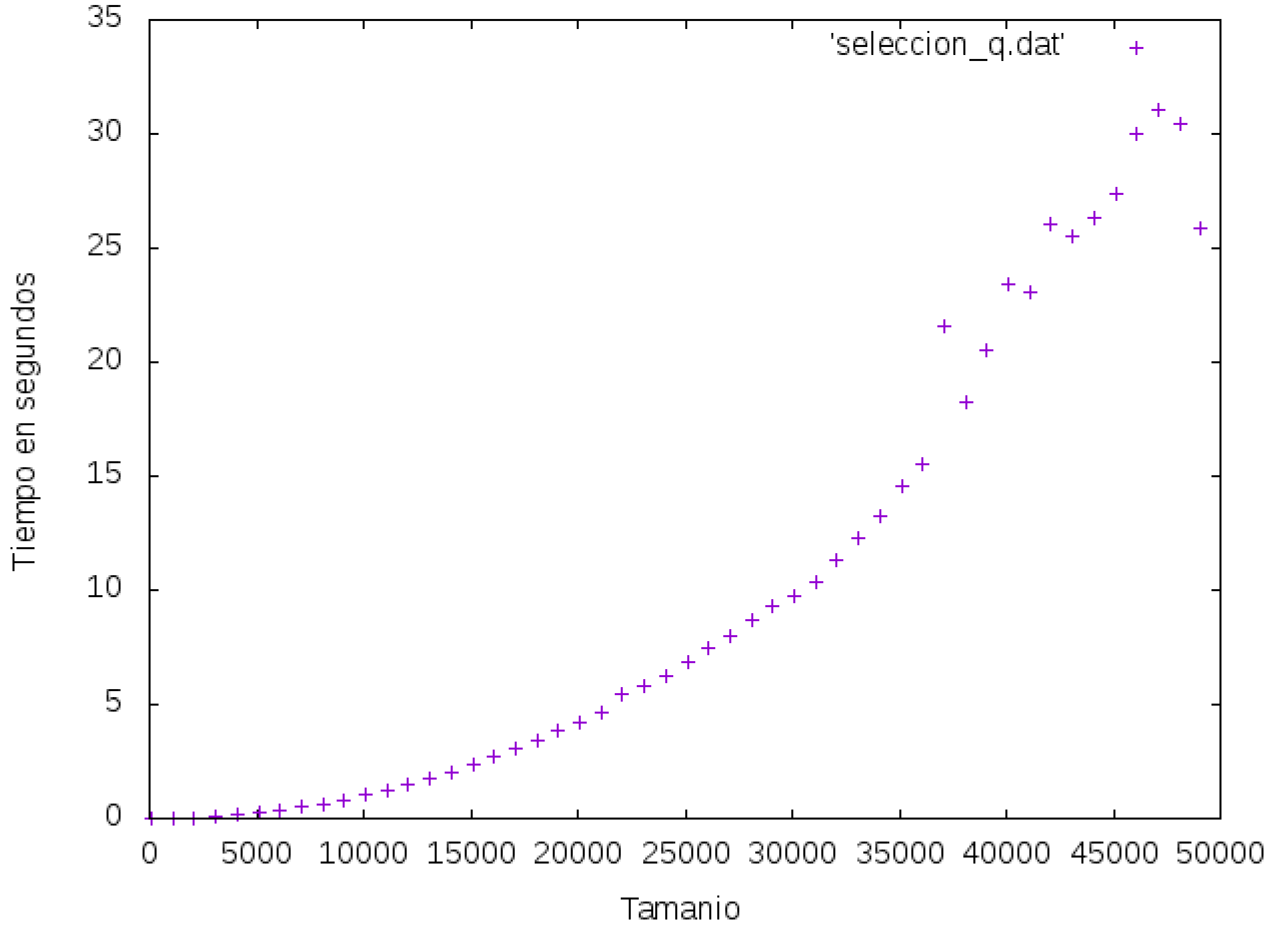
49100 32.1945



Y al hacer lo mismo pero con el Quijote:

100 0.000138143
1100 0.0114745
2100 0.0422736
3100 0.0917077
4100 0.160862
5100 0.254686
6100 0.365237
7100 0.499776
8100 0.647794
9100 0.823649
10100 1.01458
11100 1.24449
12100 1.49041
13100 1.73819
14100 2.0503
15100 2.39879
16100 2.70318
17100 3.06567
18100 3.41402
19100 3.8201
20100 4.22434
21100 4.67701
22100 5.40704
23100 5.77129
24100 6.23976

25100	6.83746
26100	7.44622
27100	8.00005
28100	8.67231
29100	9.26146
30100	9.69522
31100	10.39
32100	11.3538
33100	12.2996
34100	13.2368
35100	14.5183
36100	15.5165
37100	21.5635
38100	18.2827
39100	20.4935
40100	23.4225
41100	23.071
42100	26.0859
43100	25.535
44100	26.3359
45100	27.3854
46100	30.0201
47100	31.0798
48100	30.4303
49100	25.8621



Como podemos observar en ambos casos la gráfica es cuadrática, coincidiendo con el resultado teórico, y que no hay diferencias muy grandes entre ambos archivos.

- <u>Inserción:</u>
  - <u>Teóricamente:</u>

```
for (i = ini; i < n; i++)
{
  v = T[i];
  j = i;
  while (j > 0 && T[j-1] > v)
  {
        T[j] = T[j-1];
        j--;
  }

     T[j] = v;
 }
```

$T_{bwhile} = \sum_{v=i}^{1} O(1)$ = 1 + 1 + 1 … + 1 = i que pertenece a O(i)

$T_{iti} = T_{comp} + T_{cuerpo} + T_{inc}$ = O(1) + O(i) + O(1) = (max(1,n,1)) = O(i)

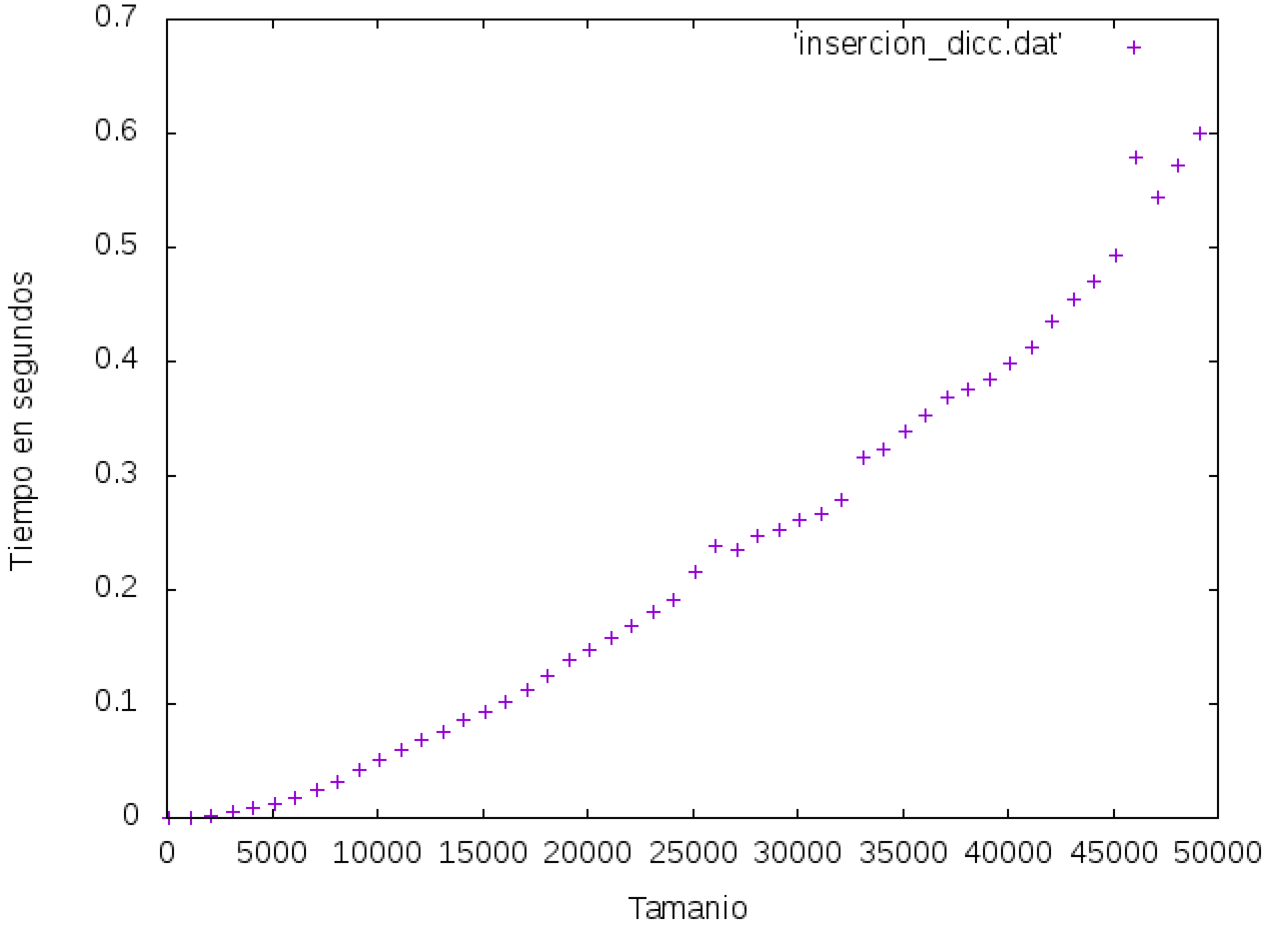
$T_{bi} = \sum_{i=1}^{n-1} i$ = $(n^2 – n)/2$ que pertenece a $O(n^2)$

- Prácticamente:

Al ordenar por inserción el diccionario:

100 1.9767e-05
1100 0.000642214
2100 0.00253043
3100 0.00448689
4100 0.0079524
5100 0.0115691
6100 0.017571
7100 0.0237795
8100 0.0322085
9100 0.0413747
10100 0.0509636
11100 0.0591867
12100 0.0676106
13100 0.0760305
14100 0.0852362
15100 0.0936719
16100 0.101373
17100 0.112431
18100 0.125433
19100 0.13803
20100 0.147363
21100 0.158271
22100 0.168957
23100 0.179827
24100 0.191706
25100 0.216232
26100 0.238865
27100 0.23465
28100 0.247909
29100 0.252302
30100 0.260853
31100 0.26737
32100 0.279271
33100 0.315597
34100 0.323058
35100 0.338329
36100 0.352265
37100 0.368083
38100 0.374822
39100 0.383832
40100 0.397436
41100 0.413125
42100 0.434828
43100 0.455143
44100 0.470559
45100 0.493026
46100 0.579817
47100 0.544353
48100 0.571335

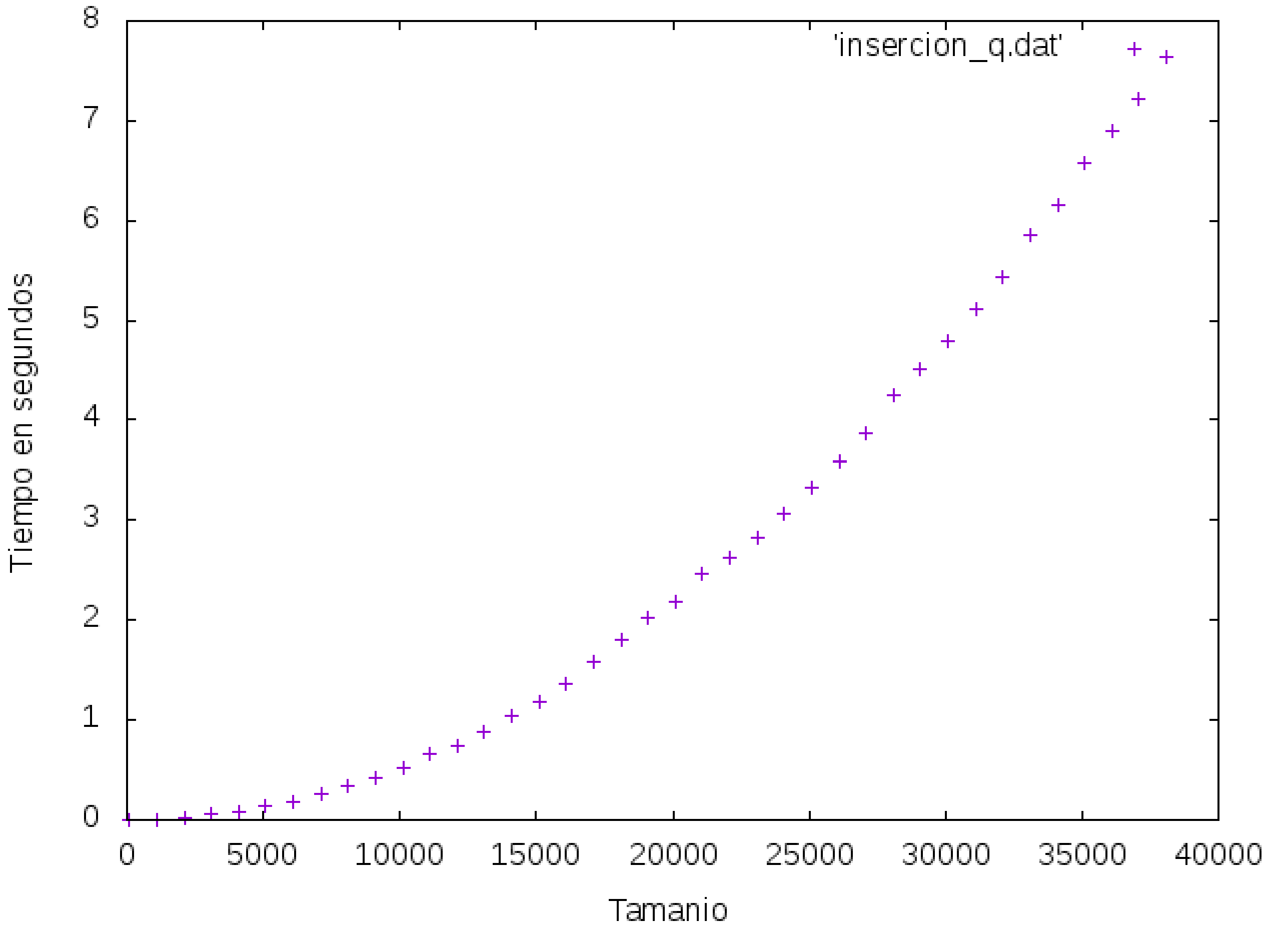
49100 0.600334



Al ordenar por inserción el Quijote:

100 0.000135508
1100 0.00716337
2100 0.0258049
3100 0.0518955
4100 0.0843086
5100 0.131013
6100 0.187156
7100 0.258193
8100 0.345478
9100 0.424896
10100 0.522748
11100 0.669798
12100 0.750614
13100 0.880857
14100 1.0384
15100 1.1781
16100 1.36433
17100 1.5789
18100 1.80802
19100 2.02095
20100 2.19353
21100 2.46737
22100 2.62296

23100 2.83114
24100 3.07405
25100 3.32867
26100 3.58074
27100 3.86964
28100 4.24099
29100 4.51512
30100 4.79585
31100 5.11325
32100 5.43773
33100 5.85024
34100 6.14627
35100 6.57786
36100 6.89472
37100 7.2272
38100 7.64878
39100 8.14205
40100 8.41609



En este caso la diferencia entre ambos archivos si se puede apreciar. Al ordenar el Quijote que está más desordenado tenemos una función cuadrática, mientras que al ordenar el diccionario que está más ordenado tiene una función casi lineal.

Conclusión

- El algoritmo de ordenación de burbuja tiene un funcionamiento muy malo cuando el conjunto a ordenar tiene un número muy grande de elementos. Esto se debe a que requiere n^2 pasos para cada n números de elementos a ser ordenados.
- El algoritmo de ordenación por selección funciona bien con una lista pequeña de elementos. Este algoritmo tiene un mal funcionamiento cuando se trata de una lista de elementos muy grande, porque al igual que la ordenación de burbuja, necesita n^2 pasos para cada n números de elementos a ser ordenados.
- El ordenamiento por inserción funciona muy bien cuando se trata de una lista muy pequeña y que ya está más o menos ordenada. Al igual que los otros dos algoritmos, cuando se trata de una lista muy grande deja de funcionar bien y tarda demasiado.