Práctica 3
Estructura de computadores

FRANCISCO NAVARRO MORALES
Universidad de Granada
27 de noviembre de 2016

Índice

1. PopCount.

El objetivo de esta primera parte es comparar distintas implementaciones de un algoritmo PopCount (cuyo objetivo es calcular el peso Hamming[1] de una lista de números) para comprobar hasta qué punto puede o no ser útil escribir ciertos programas (o partes de un programa) en ensamblador.

1.1. Primera y segunda versión.

Basándome en las transparencias vistas en clase (tr.43 y 38) realizo una primera versión (Figura 3) con un bucle for y una segunda (Figura 4) con un bucle while para ir extrayendo y acumulando bits. Los resultados obtenidos (Figura 1) indican una diferencia (favorable para la segunda versión) de unos 0.03 segundos. La principal diferencia entre estas dos implementa-

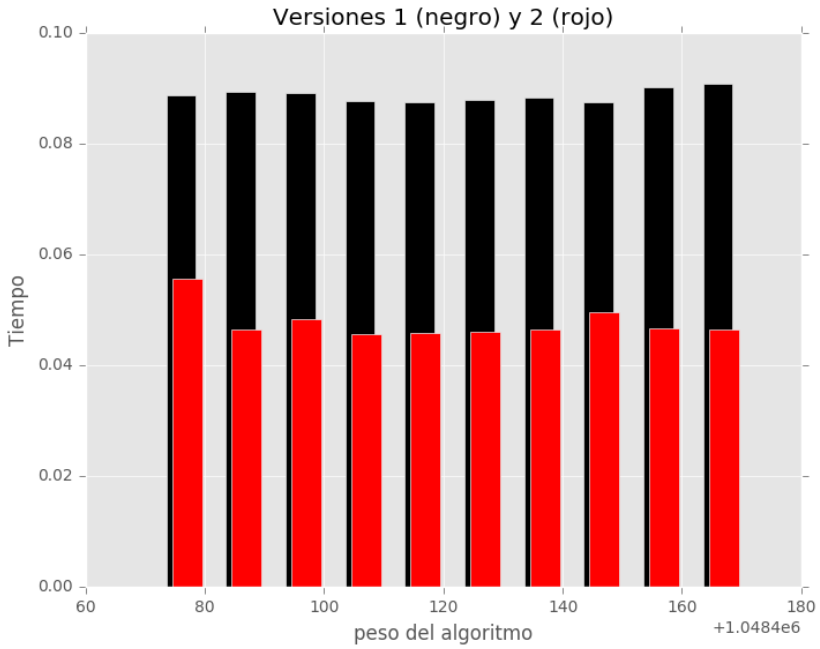

Figura 1: 1 y 2

ciones es que la primera utiliza un iterador para el bucle for, que debe ser incrementado en cada iteración del bucle y que se utiliza para comprobar si se ha terminado de procesar el número ; mientras que la segunda versión prescinde de dicho iterador y se limita a comprobar en cada iteración si el

[1]Número de unos que tiene el número en binario.

número es mayor que cero (ya que en cada iteración los bits del número son desplazados a la derecha y, por tanto, el número disminuye hasta ser cero). Es decir, que a pesar de que la diferencia es mínima (simplemente el incremento de una variable en cada iteración del bucle), ya se produce un cambio notable en los tiempos de ejecución de estas versiones.

1.2. Tercera versión.

La siguiente versión a comparar consiste en introducir un tramo (dentro del programa escrito en C) de instrucciones en ensamblador, con el objetivo de ver si realmente merece la pena introducir este tipo de instrucciones en nuestros programas. Dado que el bit desplazado acaba en el acarreo (debido a la forma en que actúa la orden SHR), lo podemos sumar de ahí mismo y nos ahorramos algunas instrucciones. Dado que ya hemos comprobado que la segunda versión es más rápida que la primera, es interesante comparar esta tercera versión (Figura 5) solo con la segunda. (Figura 2)

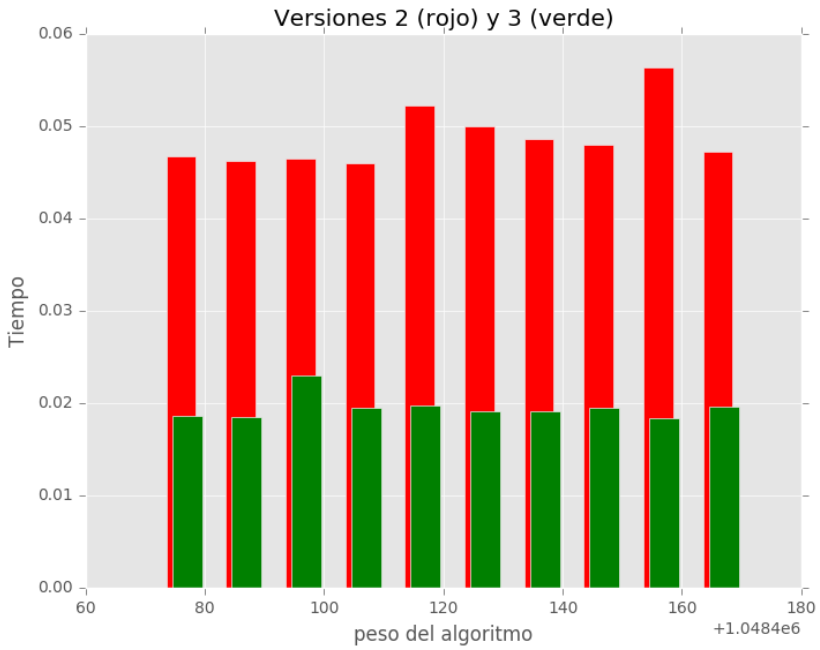

Figura 2: 2 y 3

```c
#include <stdio.h>  // para printf
#include <stdlib.h> //para exit
#define WSIZE 8*sizeof(int)

const unsigned long long SIZE = 1048576  ; // 2 elevado a 20

       unsigned long long size ;

  int pcount_for(unsigned * lista){

    long long i,x ;
    int j ;
    int result = 0 ;

    for( i = 0 ; i < size ; i++){
      x = lista[i] ;
    for( j = 0 ; j < WSIZE ; j++) {
      unsigned mask = 1 << j ;
      result += (x & mask) != 0 ;
    }
    }
    return result ;

  }

int main(int argc, char * argv[]){

  if(argc == 2)
    size = atoi(argv[1]) ;
  else size = SIZE ;

       unsigned lista[size] ;

       for(int i = 0 ; i < size ; i++)
   lista[i] = i ;

      int resultado = pcount_for(lista) ;

      printf("%d\n", resultado) ;
    }
```

primerafor.c

Figura 3: Código de la primera versión

```c
#include <stdio.h>  // para printf
#include <stdlib.h> //para exit
#define WSIZE 8*sizeof(int)

const unsigned long long SIZE = 1048576  ; // 2 elevado a 20
    const int SIZE = 4 ;
unsigned long long size;

int pcount(unsigned * lista){

    unsigned i = 0 ;
    unsigned x = 0 ;
    unsigned result = 0 ;

    for( i = 0 ; i < size ; i++){
      x = lista[i] ;
      while(x){
  result += x & 0x1 ;
  x >>= 1 ;

    }
    }
    return result ;

  }

int main(int argc, char * argv[]){

  if(argc == 2)
    size = atoi(argv[1]) ;
  else size = SIZE ;


       unsigned lista[size] ;


       for(int i = 0 ; i < size ; i++)
   lista[i] = i ;


      int resultado = pcount(lista) ;

      printf("%d\n", resultado) ;
    }
```

segundawhile.c

Figura 4: Código de la segunda versión

```c
#include <stdio.h>  // para printf
#include <stdlib.h> //para exit


const unsigned long long SIZE = 1048576  ; // 2 elevado a 20
unsigned long long size ;

  int pcount(unsigned * lista){

    unsigned i = 0 ;
    unsigned x = 0 ;
    unsigned result = 0 ;

    for( i = 0 ; i < size ; i++){
      x = lista[i] ;
      asm("\n"
    "ini3:                  \n\t"
    "shr %[x]               \n\t"
    "adc $0,  %[r]          \n\t"
    "cmp $0, %[x]           \n\t"
    "jne ini3               \n\t"

    : [r]"+r" (result)
    : [x]"r"  (x)     ) ;
    }
    return result ;
  }


int main(int argc, char * argv[]){

  if(argc == 2)
    size = atoi(argv[1]) ;
  else size = SIZE ;

       unsigned lista[size] ;

       for(int i = 0 ; i < size ; i++)
   lista[i] = i ;

      int resultado = pcount(lista) ;

      printf("%d\n", resultado) ;
    }
```

terceraADC.c

Figura 5: Código de la tercera versión

1.3. Cuarta versión.

La cuarta versión (Figura 7), consiste en aplicar la máscara 0x01010... a cada número de forma que se vayan acumulando los bits de cada byte en una nueva variable y sumar en árbol (de esta forma se cuenta un bit por byte en cada iteración y el número de iteraciones disminuye). Esta cuarta versión debería ser más rápida que las anteriores (Figura 6) y nos demostraría, de ser así, lo difícil que es superar a GCC en cuanto a optimización.

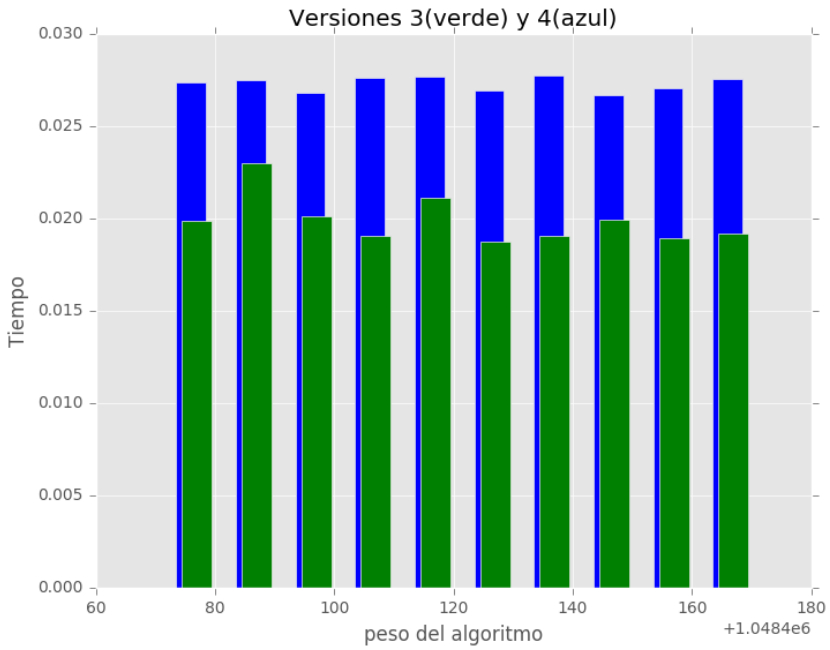

Figura 6: 1 y 2

Al contrario de lo que cabría esperar, la versión escrita en C con algunos arreglos en ASM es algo mejor que el programa (mejor pensado en ASM) escrito totalmente en esamblador.

1.4. Optimizaciones.

Antes de avanzar más, es interesante plantearse el modo de compilación de los programas escritos en C; ya que si compilamos estos aumentando las optimizaciones que genera GCC podremos obtener tiempos mucho mejores. Algunos ejemplos: En la (Figura 8), se puede apreciar que la versión 2 (solo código en C) ya supera a la 3, solo aplicando las optimizaciones (-O3). Así mismo, la Figura 9 nos muestra que, pidiendo a GCC que optimice las

```c
#include <stdio.h>
#include <stdlib.h>

const unsigned long long SIZE = 1048576  ; // 2 elevado a 20
    const int SIZE = 4 ;
unsigned long long size ;

int pcount(unsigned *lista){

  unsigned i,j;
  unsigned x = 0 ;
  unsigned result ;
  unsigned val = 0 ;

  for(i = 0 ; i < size ; i++){
    x = lista[i] ;
    for( j = 0 ; j < 8 ; j++){
          val    += x & 0x01010101 ;
          x >>= 1;
  }
    val += (val >> 16) ;
    val += (val >> 8 ) ;

    result += (val & 0xFF ) ;
    val = 0 ;
  }
  return result;              ;
}




int main(int argc, char * argv[]){

  if(argc == 2)
    size = atoi(argv[1]) ;
  else size = SIZE ;

       unsigned lista[size] ;


       for(int i = 0 ; i < size ; i++)
   lista[i] = i ;


      int resultado = pcount(lista) ;

      printf("%d\n", resultado) ;
    }
```

cuartamask.c

Figura 7: Código de la cuarta versión

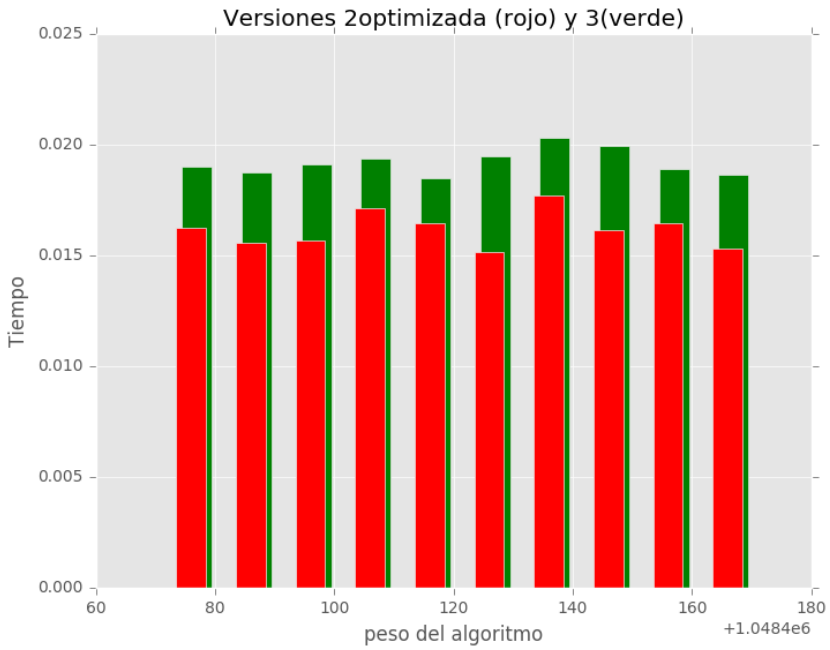

Figura 8: 2 y 3 optimizada

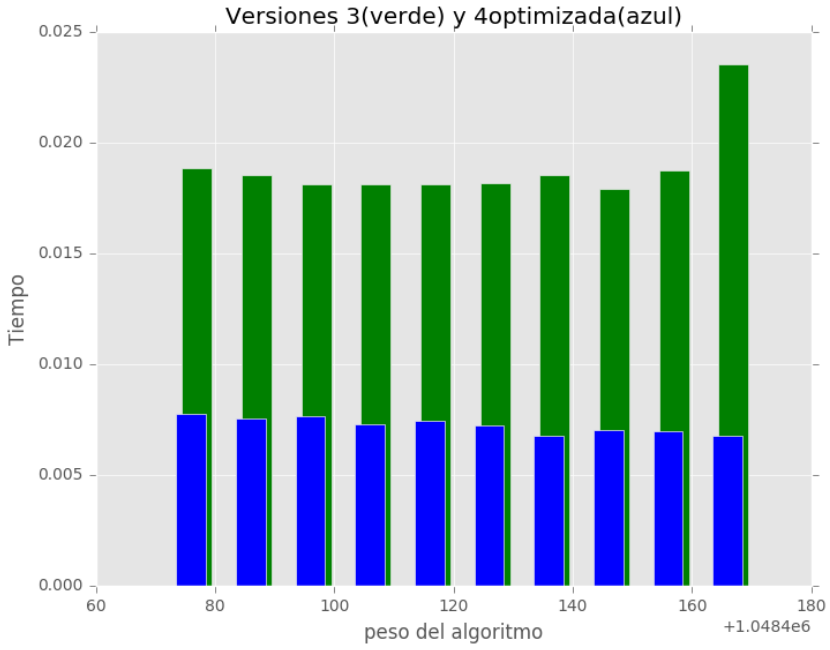

Figura 9: 3 y 4optimizada

versiones 3 y 4 (para optimizar la versión 3 hay que cambiar la etiqueta « init3 » por « 0 » o nos dará un error de compilación")

1.5. Quinta versión.

La última versión a comparar la podemos encontrar en Github, en la dirección: PopCount-SSSE Y una explicación gráfica en : explicación del algoritmo Grosso modo, podríamos decir que se trata de una versión mucho más compleja (y premeditada) que la de la versión 4, que realiza sumas de bites en paralelo y utiliza los deplazamientos para agilizar el proceso. Si lo comparamos con la versión optimizada de la cuarta implementación (La mejor hasta ahora) el resultado es el que se muestra en la figura 10.

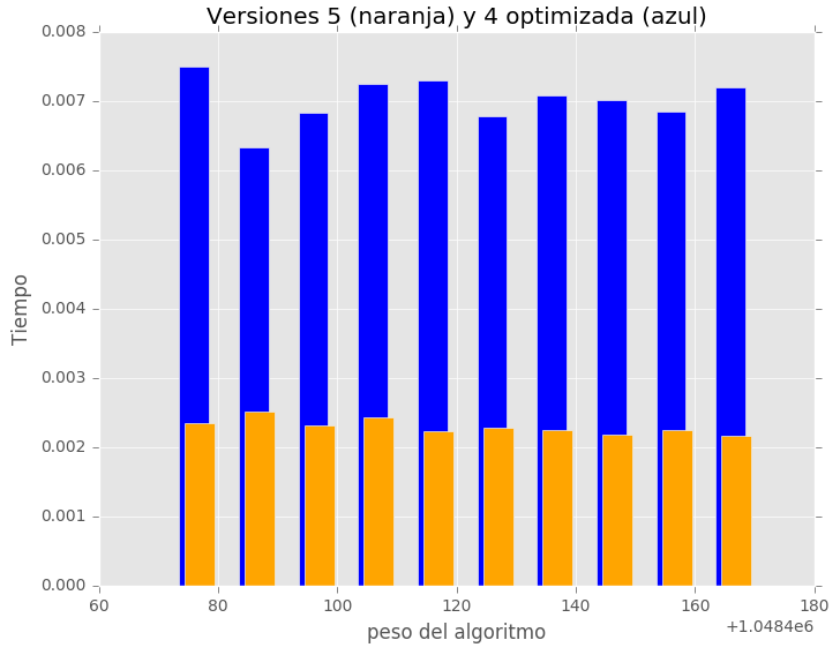

Figura 10: 4 y 5

2. Parity

En esta segunda práctica realizaremos un procedimiento similar al anterior, pero esta vez con un algoritmo que calcula la cantidad de números con paridad impar del vector.

2.1. Primera y segunda versión

Las dos primeras versiones son algo similares a las del algoritmo PopCount, con un bucle for y un bucle while respectivamente.

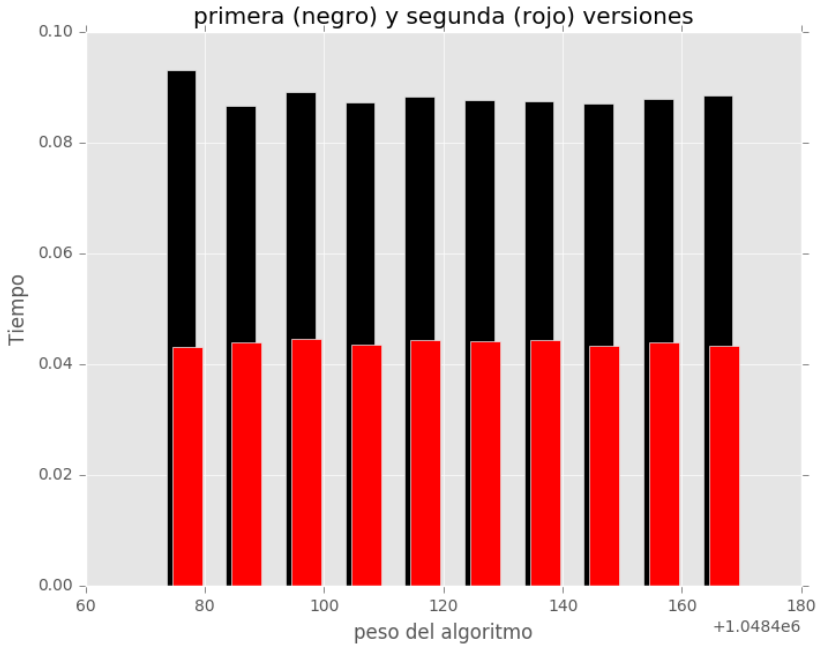

Figura 11: 1 y 2

Se puede apreciar fácilmente una cierta mejoría al usar el bucle while por las mismas razones que en el caso del PopCount.

No es demasiado interesante comparar los distintos niveles de optimización puesto que, -O1 y -O2 dan resultados muy similares; bastante mejores, eso sí, que -O0.

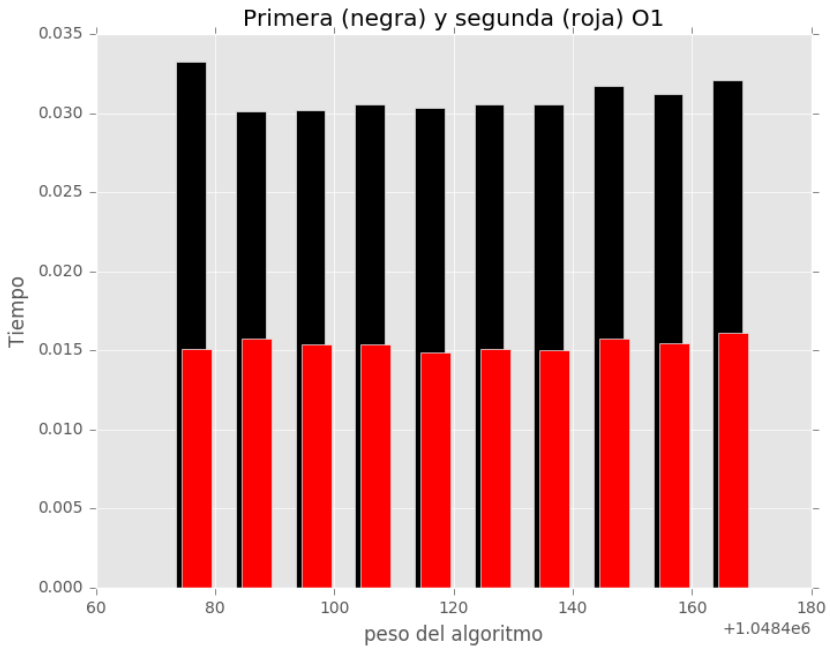

Figura 12: 1 y 2

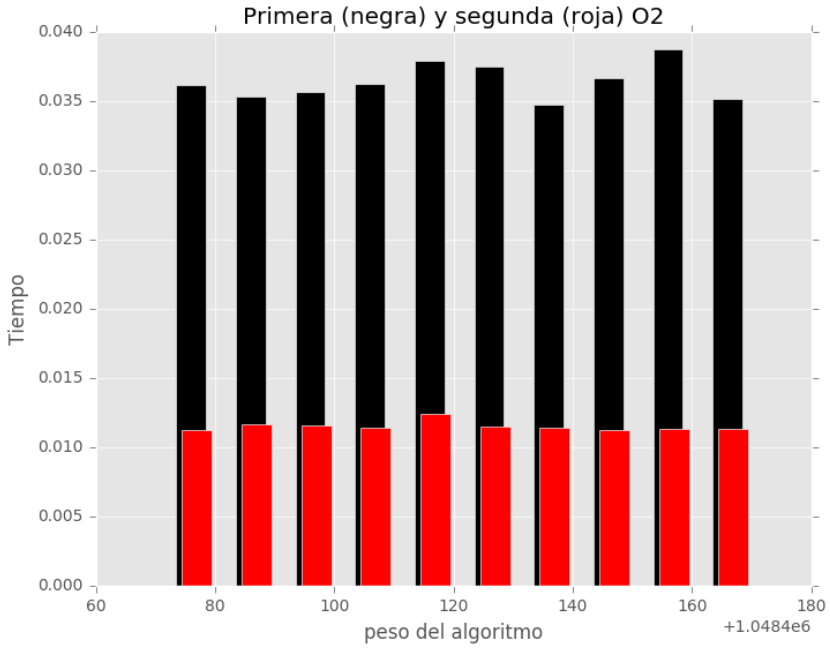

Figura 13: 1 y 2

```c
#include <stdio.h>  // para printf
#include <stdlib.h> //para exit
#define WSIZE 8*sizeof(int)

const unsigned long long SIZE = 1048576  ; // 2 elevado a 20

       unsigned long long size ;

//unsigned lista[]={0x80000000, 0x001000000, 0x00000800,  0
    x00000001 } ;
////////////////// (1 uno,  1 uno, 1 uno, 1 uno )



  int pcount(unsigned * lista){

    long long i,x ;
    int j ;
    unsigned result = 0, par = 0  ;
    for( i = 0 ; i < size ; i++){
      x = lista[i] ;
      par = 0 ;
      for( j = 0 ; j < WSIZE ; j++) {
  unsigned mask = 1 << j ;
  par ^= (x & mask) != 0 ;
      }
      result += par ;
    }
    return result ;

  }

int main(int argc, char * argv[]){

  if(argc == 2)
    size = atoi(argv[1]) ;
  else size = SIZE ;

       unsigned lista[size] ;

       for(int i = 0 ; i < size ; i+=4){
   lista[i]   = 0x80000000  ;
         lista[i+1] = 0x00100000  ;
   lista[i+2] = 0x00000800  ;
   lista[i+3] = 0x00000001  ;
       }
      int resultado = pcount(lista) ;

      printf("%d\n", resultado) ;
    }
```

pprimera.c

Figura 14: Código de la primera versión

```c
#include <stdio.h>  // para printf
#include <stdlib.h> //para exit
#define WSIZE 8*sizeof(int)

const unsigned long long SIZE = 1048576  ; // 2 elevado a 20
    const int SIZE = 4 ;
unsigned long long size;
//unsigned lista[]={0x80000000, 0x0010000, 0x00000800, 0
    x00000001 } ;
////////////////// (1 uno,  1 uno, 1 uno, 1 uno )



int pcount_for(unsigned * lista){

    unsigned i = 0 ;
    unsigned x = 0 ;
    unsigned result = 0 ;

    for( i = 0 ; i < size ; i++){
      x = lista[i] ;
      unsigned par = 0 ;
      while(x){
  par ^= x & 0x1 ;
  x >>= 1 ;
      } result += par ;
    }
    return result ;

  }

int main(int argc, char * argv[]){

  if(argc == 2)
    size = atoi(argv[1]) ;
  else size = SIZE ;


       unsigned lista[size] ;


       for(int i = 0 ; i < size ; i+=4){
   lista[i]   = 0x80000000  ;
         lista[i+1] = 0x00100000  ;
   lista[i+2] = 0x00000800  ;
   lista[i+3] = 0x00000001  ;
       }

      int resultado = pcount_for(lista) ;

      printf("%d\n", resultado) ;
    }
```

psegunda.c

Figura 15: Código de la segunda versión

2.2. Tercera versión

Dado que lo que estamos contando es la cantidad de números con paridad impar podemos ahorrarlos la máscara para separar el bit menos significativo (que hacíamos en el bucle while en la versión 2) y limitarnos a aplicar dicha máscara al final del bucle, a la hora de sumar el resultado (cero o uno) al acumulador resultado. Es decir, lo que vamos haciendo es hacer la operación xor del número completo en cada iteración del bucle while (sin la máscara) pero en realidad lo que nos va a valer es sólo el bit menos significativo; los demás bits no nos interesan pero no importa, porque se van a perder trás el bucle while, al acumular los resultados. Aún sabiendo esto, las diferencias de tiempo entre la segunda y la tercera versión son mínimas, en los distintos grados de optimización quizá sea -O2 cuando más diferencia puedo apreciar.

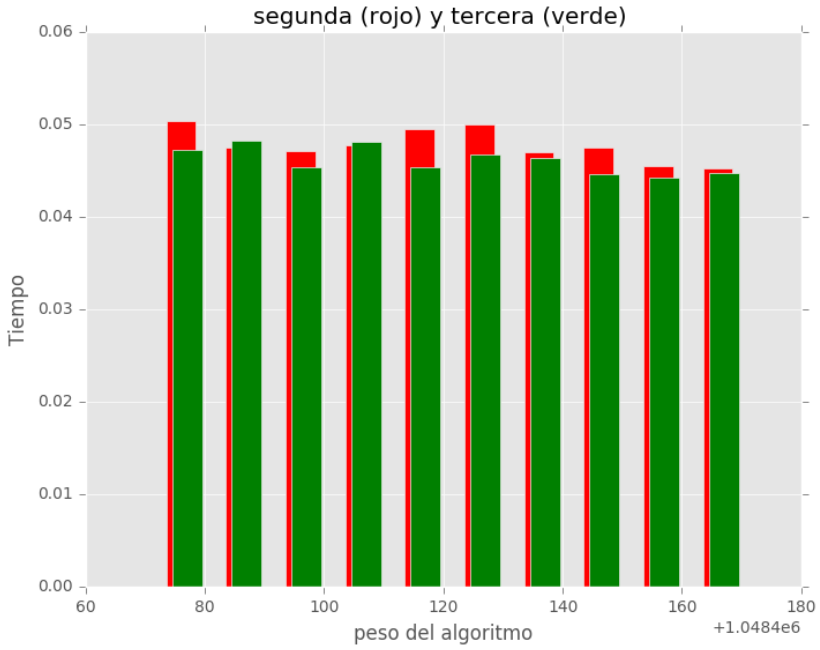

Figura 16: 1 y 2

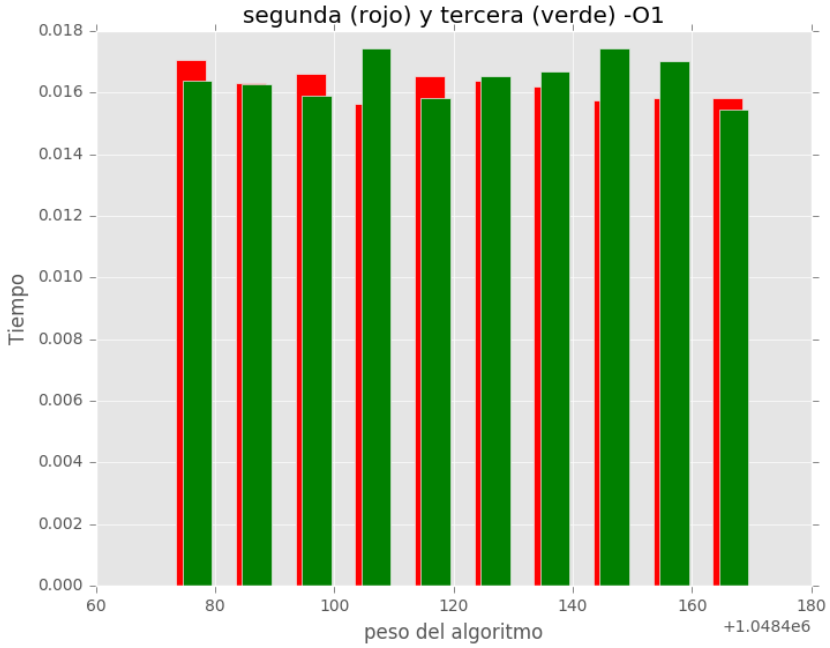

Figura 17: 1 y 2

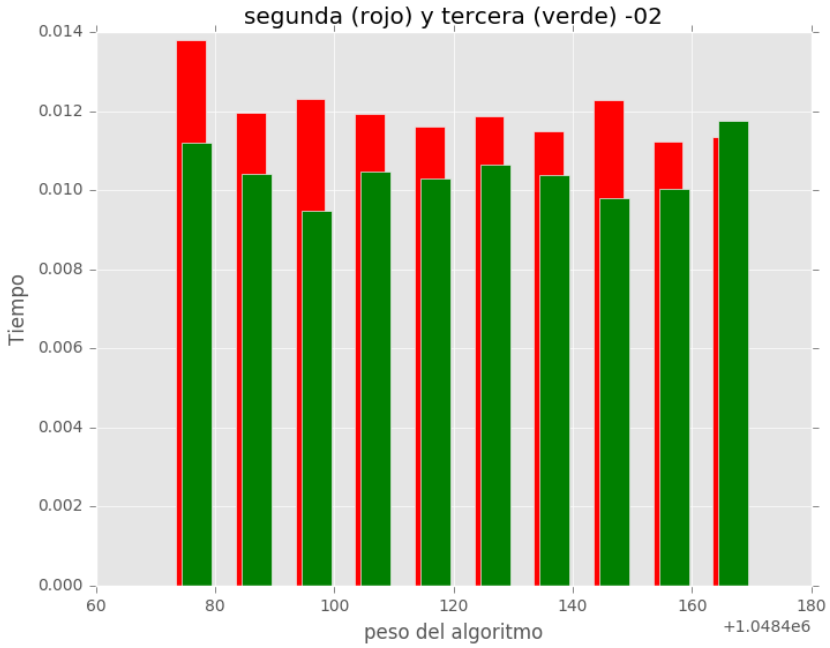

Figura 18: 1 y 2

```c
#include <stdio.h>  // para printf
#include <stdlib.h> //para exit


const unsigned long long SIZE = 1048576  ; // 2 elevado a 20
unsigned long long size ;
//unsigned lista[]={0x80000000, 0x00100000, 0x00000800, 0
    x00000001 } ;
////////////////// (1 uno,  1 uno, 1 uno, 1 uno )



  int pcount_for(unsigned * lista){

    unsigned i = 0 ;
    unsigned x = 0 ;
    unsigned result = 0 ;
    unsigned par = 0 ;

    for( i = 0 ; i < size ; i++){
      x = lista[i] ;
      par = 0 ;
      while(x){
  par ^= x ;
  x >>= 1 ;
      } result += par & 0x1 ;
    }
    return result ;

  }



int main(int argc, char * argv[]){

  if(argc == 2)
    size = atoi(argv[1]) ;
  else size = SIZE ;

       unsigned lista[size] ;


       for(int i = 0 ; i < size ; i+=4){
   lista[i]   = 0x80000000  ;
         lista[i+1] = 0x00100000  ;
   lista[i+2] = 0x00000800  ;
   lista[i+3] = 0x00000001  ;
       }

      int resultado = pcount_for(lista) ;

      printf("%d\n", resultado) ;
    }
```

ptercera.c

Figura 19: Código de la tercera versión (parity)

2.3. Cuarta versión

Esta vez, al igual que hicimos en el PopCount, vamos a intentar mejorar la implementación añadiendo un bloque ASM que hará uso de la orden SHR al igual que hicimos en el bloque ASM del algoritmo PopCount. Es interesante ver que, al compilar, el código obtenido en O1 es prácticamente el mismo que el de la versión anterior; sin embargo, sin optimización y con -O2 esta versión supera a la tercera.

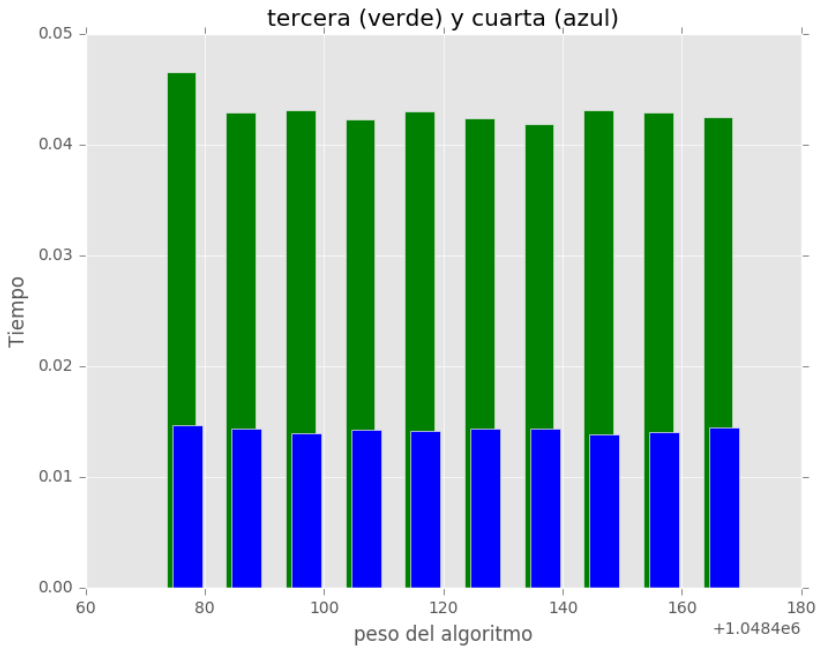

Figura 20: 1 y 2

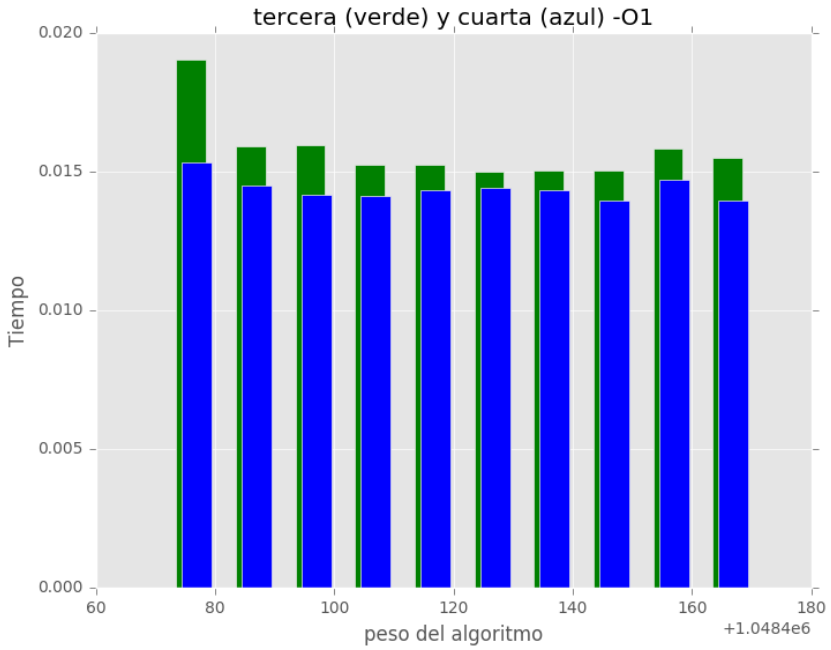

Figura 21: 1 y 2

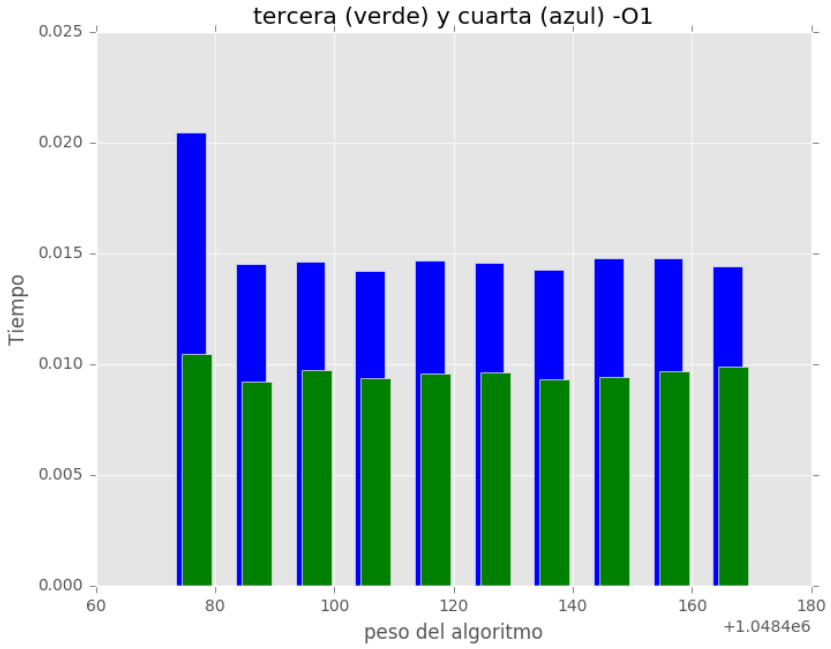

Figura 22: 1 y 2

```c
#include <stdio.h>  // para printf
#include <stdlib.h> //para exit

const unsigned long long SIZE = 1048576  ; // 2 elevado a 20
unsigned long long size ;

  int pcount(unsigned * lista){

    unsigned i = 0 ;
    unsigned x = 0 ;
    unsigned result = 0 ;
    unsigned par = 0 ;

    for( i = 0 ; i < size ; i++){
      x = lista[i] ;
      par = 0 ;
   asm("\n"
  "ini3%=:                        \n\t"
  "xor %[x],%[par]              \n\t"
  "shr %[x]                     \n\t"
  "test %[x], %[x]              \n\t"
  "jnz ini3%=                     \n\t"
  "and $1, %[par]               \n\t"
  : [par]"+r" (par)
  : [x]"r" (x)
  ) ;

    result += par ;
    }
    return result ;

  }



int main(int argc, char * argv[]){

  if(argc == 2)
    size = atoi(argv[1]) ;
  else size = SIZE ;

       unsigned lista[size] ;


       for(int i = 0 ; i < size ; i+=4){
   lista[i]   = 0x80000000  ;
         lista[i+1] = 0x00100000  ;
   lista[i+2] = 0x00000800  ;
   lista[i+3] = 0x00000001  ;
       }

      int resultado = pcount(lista) ;

      printf("%d\n", resultado);
   }
```

pcuarta.c

Figura 23: Código de la cuarta versión (parity)

2.4. Quinta versión

Al igual que hicimos con el algoritmo PopCount, podemos realizar una versión que realice la comprobación de paridad en árbol; es decir: dado que nuestro objetivo es saber si el número de dígitos uno es impar, podemos ir haciendo la operación xor del número consigo mismo (desplazándolo un poco cada vez) de forma que este se vaya partiendo en dos mitades las cuales se van sumando (suma exclusiva xor). Dado que la suma exclusiva devuelve un 1 si los dígitos sumados eran distintos y un 0 si eran iguales, cada vez que se sumen dos unos, estos se anularán (se volverán un cero), lo cual nos interesa puesto que queremos saber si hay un número impar de unos, pero no cuantos. Así, dado que cada vez el número se divide en partes más que pequeñas que se van sumando (xor), nos encontraremos en que al final tenemos en el bit menos significativo 1 o 0 dependiendo de si el número de unos iniciales era impar o no. El resto de los dígitos no tienen importancia. Aunque pudiera parecer que el algoritmo es mucho mejor que la versión anterior, si no planificamos bien como implementarlo y copilamos con O0, los resultados son incluso peores que los de la cuarta versión. No obstante, basta compilar con -O1 u -O2 para ver la mejoría.

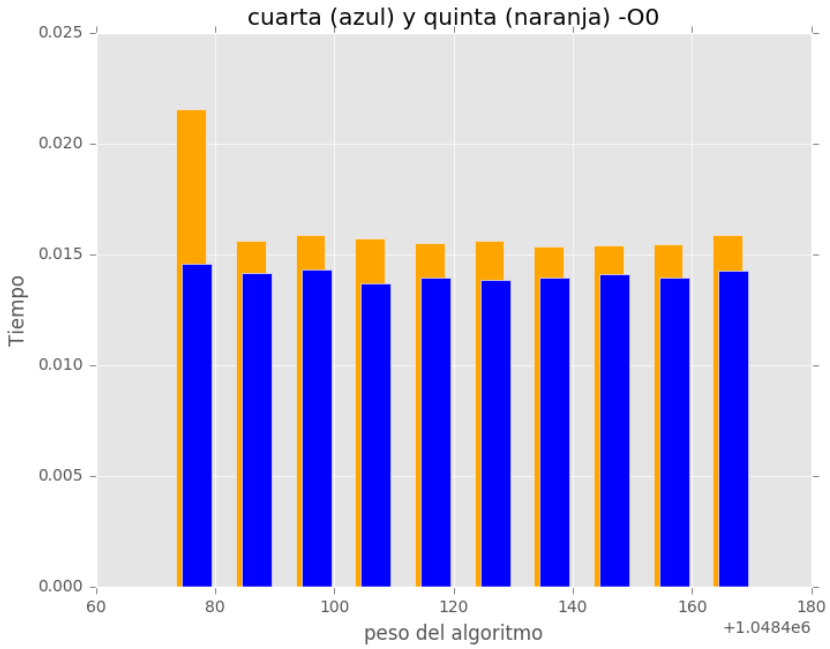

Figura 24: 4 y 5

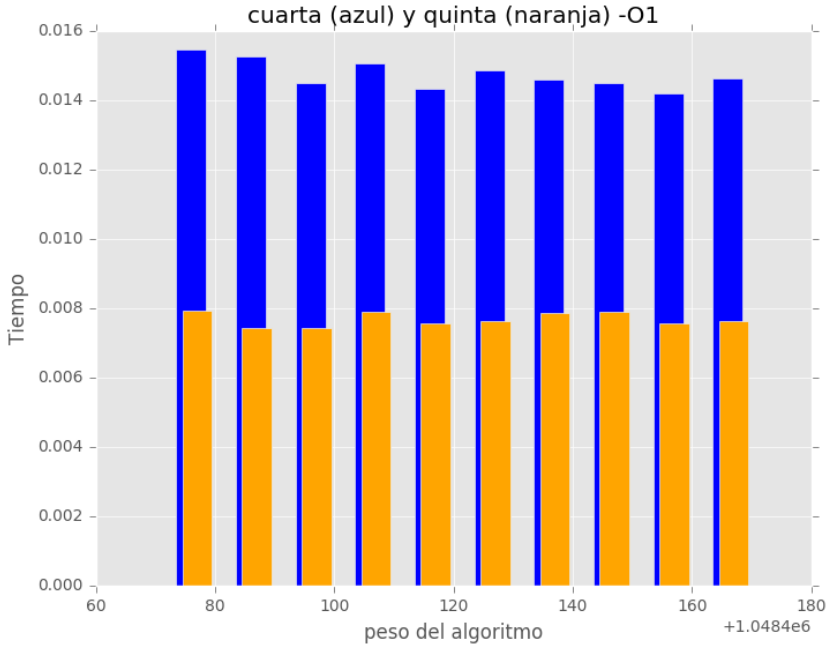

Figura 25: 4 y 5

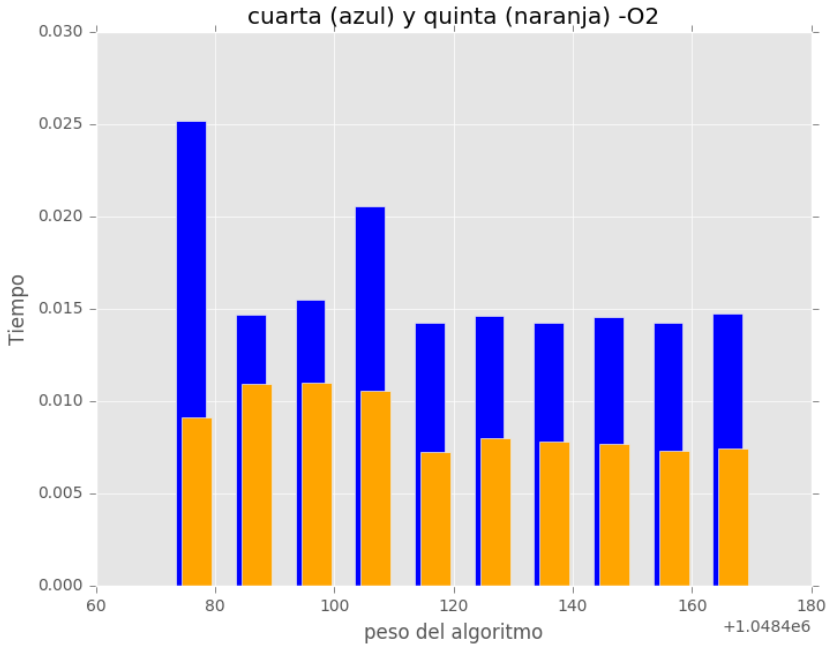

Figura 26: 4 y 5

```c
#include <stdio.h>  // para printf
#include <stdlib.h> //para exit

const unsigned long long SIZE = 1048576  ; // 2 elevado a 20
unsigned long long size ;

  int pcount(unsigned * lista){

    unsigned i = 0, j = 0 ;
    unsigned x = 0 ;
    unsigned result = 0 ;

    for( i = 0 ; i < size ; i++){
      x = lista[i] ;
      for( j = 16 ; j >= 1 ; j=j/2)
  x ^= (x>>j) ;
      result += (x&0x1) ;
    }
    return result ;

  }



int main(int argc, char * argv[]){

  if(argc == 2)
    size = atoi(argv[1]) ;
  else size = SIZE ;

       unsigned lista[size] ;


       for(int i = 0 ; i < size ; i+=4){
   lista[i]   = 0x80000000  ;
         lista[i+1] = 0x00100000  ;
   lista[i+2] = 0x00000800  ;
   lista[i+3] = 0x00000001  ;
       }

      int resultado = pcount(lista) ;

      printf("%d\n", resultado) ;
    }
```

pquinta.c

Figura 27: Código de la quinta versión (parity)

2.5. Sexta versión

Una última versión, otra vez en ensamblador, nos permite realizar lo mismo que hemos hecho en la versión anterior pero cambiando el bucle que servía para los desplazamientos (que iba de j=16 a j=1 dividiendo cada vez j entre 2) por instrucciones que aprovechan la posibilidad de utilizar "subregistros"que componen a otros más grandes, para no necesitar realizar ningún desplazamiento. Por ejemplo, primero se utiliza el registro edx para almacenar x, entonces desplazamos 16bits x y hacemos la suma exclusiva con edx pero después, en lugar de desplazar x otros 8 bits, aprovechamos que edx está compuesto por dos registros de 8 bits, dh y dl, los cuales podemos sumar (xor) ahorrándonos un desplazamiento.También es interesante el uso que podemos darle a setpo, una instrucción que asigna a un registro de tamaño 1 Byte, como en este caso, dl, el byte 1 (representación del número 1) si el bit de paridad está activo, lo cual permite comprobar la paridad de un byte completo directamente sin necesidad de dividirlo en 4, 2 y 1 bits cómo hacíamos antes. Los resultados, por supuesto, son mucho mejores.

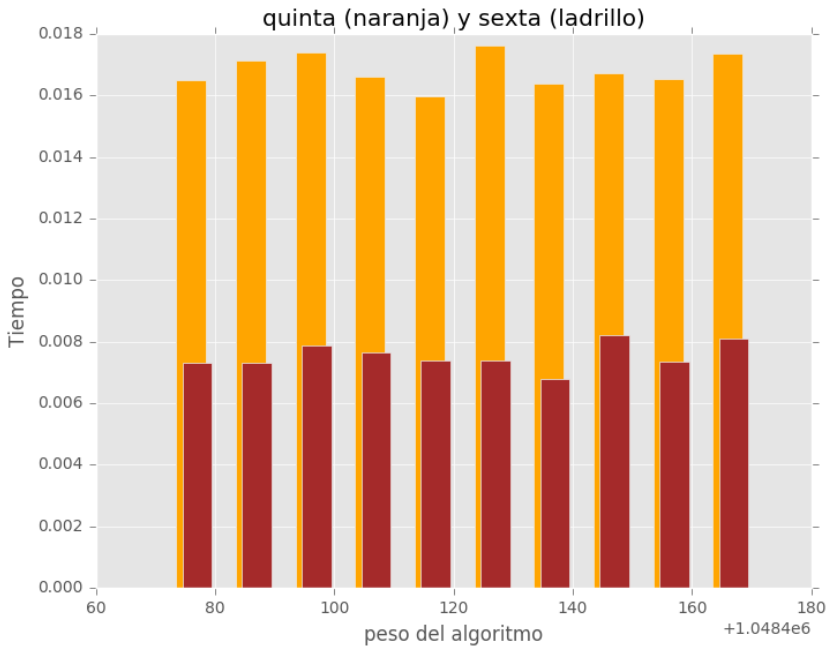

Figura 28: 5y6

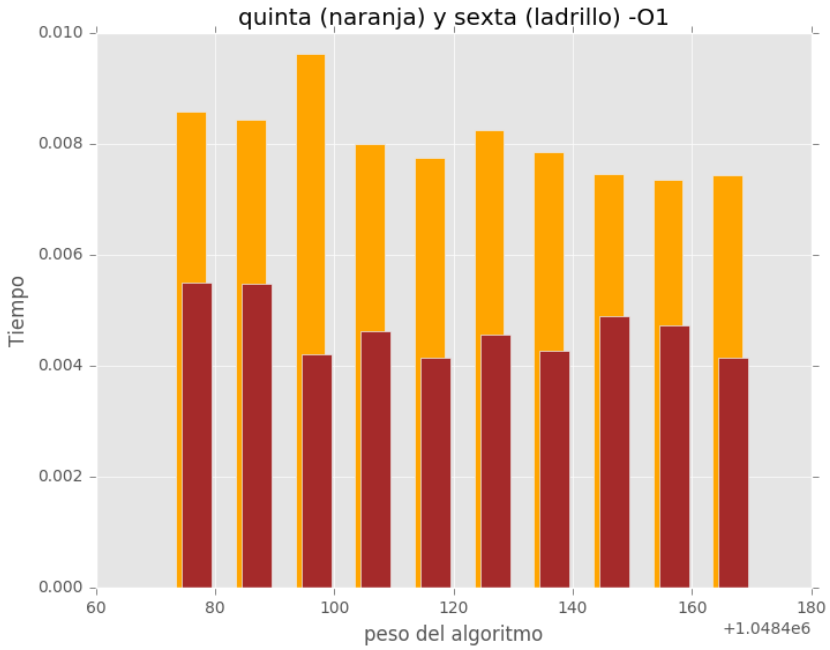

Figura 29: 5y6

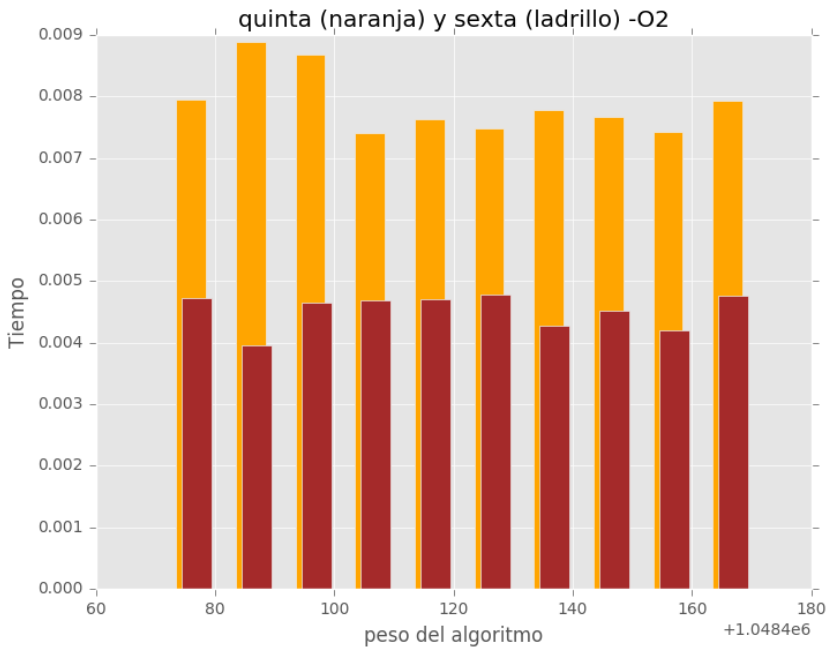

Figura 30: 5y6

```c
#include <stdio.h>  // para printf
#include <stdlib.h> //para exit

const unsigned long long SIZE = 1048576  ; // 2 elevado a 20
unsigned long long size ;

  int pcount(unsigned * lista){

    unsigned i = 0, j = 0 ;
    unsigned x = 0 ;
    unsigned result = 0 ;

    for( i = 0 ; i < size ; i++){
      x = lista[i] ;
      for( j = 16 ; j >= 1 ; j=j/2)
  x ^= (x>>j) ;
      result += (x&0x1) ;
    }
    return result ;

  }



int main(int argc, char * argv[]){

  if(argc == 2)
    size = atoi(argv[1]) ;
  else size = SIZE ;

       unsigned lista[size] ;


       for(int i = 0 ; i < size ; i+=4){
   lista[i]   = 0x80000000  ;
         lista[i+1] = 0x00100000  ;
   lista[i+2] = 0x00000800  ;
   lista[i+3] = 0x00000001  ;
       }

      int resultado = pcount(lista) ;

      printf("%d\n", resultado) ;
    }
```

pquinta.c

Figura 31: Código de la sexta versión (parity)

A. Conclusiones.

En conclusión, el aprendizaje obtenido de esta práctica bien podría ser que es muy complicado superar al compilador de C en lo que a optimización se refiere, pero es posible. He podido comprobar que, por lo general, resulta absurdo plantear un problema (por pequeño que sea) como un programa completo en ensamblador, mientras que sí que resulta práctico (sólo en ocasiones y si se tiene muy claro el objetivo y el modo de llegar a este) añadir algunas funciones en ensamblador o incluso instrucciones ASM in-line. También he podido comprobar de forma gráfica hasta qué punto puede mejorar el rendimiento de un programa en función de los niveles de optimización utilizados al compilarlo, así como ver algunos de los mecanismos que utiliza gcc para mejorar las implementaciones de nuestros algoritmos (por ejemplo, desenrollar bucles).

B. Script utilizado para las comparaciones

Para realizar la práctica se me ocurrió escribir un script en Python (ya que es un lenguaje muy versátil y relativamente sencillo de entender y/o aprender) que me permitiera medir los tiempos de dos implementaciones de un mismo programa y obtener una gráfica comparándolos. El script está pensado para programas que acepten un argumento que será el peso del problema (en este caso, el tamaño del vector) y lleva a cabo 10 ejecuciones de cada programa con pesos muy similares para poder mostrar variaciones puntuales. A la hora de insertar un gráfico en el documento me he decantado por obtener dos o tres con el script y luego elegir el que más uniforme me parecía. También he intentado utilizar el script para generar los gráficos en momentos en los que mi ordenador estaba lo menos sobrecargado posible.

el script en cuestión:

```
#!/usr/bin/python
import sys
import optparse
import subprocess
import time
import os
import matplotlib
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import style
style.use('ggplot')
parser = optparse.OptionParser()
parser.add_option('-f', '--first', dest='first', help='first program')
parser.add_option('-s', '--second', dest='second', help='second program')
parser.add_option('-F', '--colorFirst', dest='color1', help='first color')
parser.add_option('-S', '--colorSecond', dest='color2', help='second color
    ')
parser.add_option('-n', '--name', dest='name', help='output file name')
(options, args) = parser.parse_args()
if options.first is None:
  options.first = raw_input('Enter first program path: ')
if options.second is None:
  options.second = raw_input('Enter second program path: ')
titulo = raw_input('Graph title?')
if options.name is None:
  options.name = "grafico"
times1 = []
times2 = []
ind = []
ind2 = []
i = 1048476
while i < 1048576:
  start_time = time.time()
  os.system("./"+options.first + " %d > /dev/null" % (i))
  times1.append( (time.time() - start_time))

  start_time = time.time()
  os.system("./"+options.second + " %d > /dev/null" % (i))
  times2.append( (time.time() - start_time))
  ind.append(i)
  ind2.append(i+1)
  i = i+10
plt.bar(ind,times1,5, align='center',color=options.color1)
plt.bar(ind2,times2,5, align='center',color=options.color2)
plt.title(titulo)
plt.ylabel('Tiempo')
plt.xlabel('peso del algoritmo')
plt.savefig(options.name+'.png',bbox_inches='tight')
```

compare.py