DATA BUS

Número 6 Segunda Epoca
Septiembre 1.994

EDITA

ASOCIACION JUVENIL PARA LA INFORMATICA VIGUESA

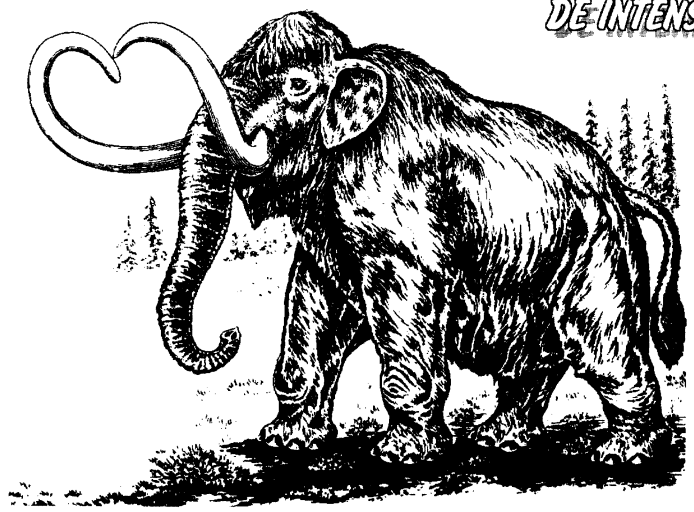
AJIV

G.D.I.

GRUPO DE DESARROLLO INFORMATICO

IMAGEN

TRANSFORMACIONES DE INTENSIDAD



ALGORITMOS

ALGORITMOS AVANZADOS DE CLASIFICACION ALFABETICA

HISTORIA DE LA INFORMATICA

PARTE IX

DATA BUS

Número 6

Segunda época

Septiembre de 1.994

DIRECTOR

Jesús Cea Avión

REDACTORES

Jesús Cea Avión

Nacho Agulló Sousa

Jose Manuel Suárez Pousa

COLABORADORES

Francisco Bellas Aláez

David Martínez Oliveira

DISEÑO Y MAQUETACION

Jesús Cea Avión

ASOCIACION JUVENIL PARA LA INFORMATICA VIGUESA

AJIV

C/Caldas de Reis, 12-6º Izq.
36209 VIGO
Telf: (986) 23 18 35

Depósito legal
VG-87-90

¡¡¡ ATENCION !!!

Si deseas colaborar con nosotros en *DATA BUS* aportando ideas, críticas, artículos, etc., no tienes más que hacernos llegar tu trabajo. Para ello puedes enviárnoslo a la dirección de la asociación, o bien ponerte en contacto con la persona que te suministró este ejemplar.

¡¡¡ CONTAMOS CON VOSOTROS !!!

INDICE

EDITORIAL

¡¡Por Fin!!. ¡Por fin tenéis otro número de ***DATA BUS*** en vuestras manos, después de una larga ausencia!. Desde luego está visto que cada vez intentamos fijar una periodicidad (no perioricidad, como se escribía en el último número) para ***DATA BUS*** la rompemos ya al número siguiente...

Como siempre, tenemos excusas: retraso en la entrega de los artículos, que este curso ha sido realmente matador, la creación del ***G.D.I.***, cambio de equipo informático, proyectos personales, y un largo etcétera...

Y considerando que el ***G.D.I.*** sigue funcionando, que dentro de unos pocos días empieza otro curso probablemente peor que el anterior y que sigo trabajando en proyectos personales... Bueno, intentaremos que el próximo número salga cuando debe, es decir, en Enero. Pero no podemos prometer nada...

Nos ayudaría, claro, que los lectores colaboráseis más enviando artículos, comentarios, ideas e, incluso, críticas. Lo importante es hacernos "notar" vuestra presencia permanente y fiel. Necesitamos vuestra realimentación para seguir en la brecha. ¡¡¡ No nos falléis !!!.

Cambiando de tema, en este número tenéis un segundo artículo de la serie de procesado de imagen, así como la segunda y última parte (probablemente) de la serie de algoritmos de clasificación. Las secciones habituales se mantienen, naturalmente: algoritmos, noticias, historia de la informática y la precisión ante todo.

También seguimos publicando pequeños listados en 'C' y esperando a que nos hagáis llegar vuestras opiniones al respecto.

Por cierto, nos ha sorprendido muy gratamente la calurosa acogida que ha tenido la sección de *"la precisión ante todo"*. ¡¡Quien lo hubiera imaginado!!. Desde luego, ha sido una auténtica sorpresa. Ni que decir tiene que mantendremos dicha sección de forma indefinida. El problema ahora es encontrar números "transcendentales" para publicar. ¿El término 1000 de la serie de Fibonacci? ¿El primer número primo de 1500 cifras? ¿La dimensión fractal de una tarta de manzana hecha en casa? ¿La distancia media de la Tierra al Sol en micrómetros? Aceptamos sugerencias...

Bueno, nada más por el momento. Espero que disfrutéis de este número de ***DATA BUS***. Nos veremos en Enero.

El Director

Una somera introducción al

Procesado de Imagen (II)

Transformaciones de intensidad

Francisco Bellas Aláez

David Martínez Oliveira

Introducción

En esta segunda entrega de nuestra serie sobre procesado digital de imagen vamos a describir diversas técnicas de ensalzado, orientadas principalmente a modificaciones de intensidad de los pixels de cada imagen. Todas los operaciones de modificación de la imagen se realizan sobre la memoria del ordenador, utilizando las técnicas descritas en el artículo anterior para poder representar los resultados en nuestros monitores y así comprobar los maravillosos resultados que podemos obtener.

Describiremos técnicas para la corrección de brillo y contraste, las cuales se relacionan directamente con operaciones sobre pixels y modificación de su intensidades. Se realizarán a lo largo del artículo alusiones a propiedades estadísticas, de las señales e intentaremos en lo posible dar una explicación práctica de lo que representa cada parámetro que introduzcamos.

Probabilidades e histograma

Todos sabemos lo que es una probabilidad. Frases como *"es probable que vaya a llover"* o *"probablemente no esté en casa"* son habituales en nuestras vidas. En el procesado de señal, la teoría de probabilidad se hace indispensable para muchas de las aplicaciones más interesantes, como puede ser la compresión, o el tema que trataremos en este artículo.

En procesado de señal nos interesaremos por las probabilidades de intensidades, es decir, queremos conocer que probabilidad tenemos de que un pixel sea de color negro o sea gris o sea blanco, o sea (Oh!, mar), de qué color es. La definición clásica de probabilidad, la que nos enseñan en el colegio, es la que vamos a utilizar. Si una caja tiene 3 bolas blanca y 4 negras ¿cuál es la probabilidad de que saquemos una bola negra? Pues será los casos favorables 4 dividido por los casos posibles, es decir 7. La probabilidad sería 4/7.

En nuestro caso nos interesa conocer cual es la probabilidad de obtener un punto de color gris claro

(caracterizado, por ejemplo por un valor 214) de entre todos de la pantalla, contaremos las veces que aparece el color 214 y lo dividiremos por el número total de puntos de que disponemos. Un histograma es un gráfico en el que se representa las ocurrencias de los elementos de un conjunto en un determinado espacio de muestreo. Nuestro histograma indicará las apariciones de cada color en la imagen que estemos procesando en ese momento. En la práctica no se suele dividir por el número total puntos, esta operación no es otra cosa que una normalización para obtener valores entre 0 y 1, lo que implica la utilización de números reales los cuales requieren un mayor gasto computacional para realizar cualquiera de la operaciones básicas (suma, producto), además de requerir más memoria para su almacenamiento.

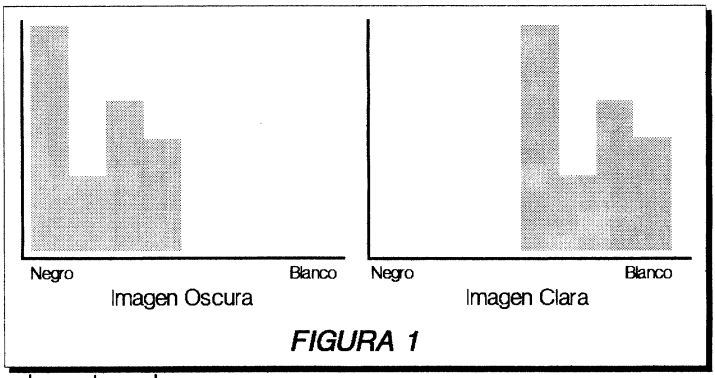


FIGURA 1

El histograma proporciona una información valiosísima sobre los parámetros de brillo y contraste de la imagen. Una imagen oscura presentará en su histograma valores altos para los colores más oscuros, en nuestro sistema el 0 representa el color negro y el 255 (trabajamos con 8 bits 2^8=256) el blanco. Una imagen muy clara presentará un histograma con valores altos para los colores más próximos al blanco (próximos al 255) pues será los que aparezcan con mayor frecuencia, con mayor probabilidad (ver figura 1). Una imagen con poco contraste presentará un histograma con su valores más concentrados en una cierta región de la gráfica, si los valores aparecen distribuidos a lo largo de toda la gráfica entonces el contraste de la imagen será mayor (ver figura 2).

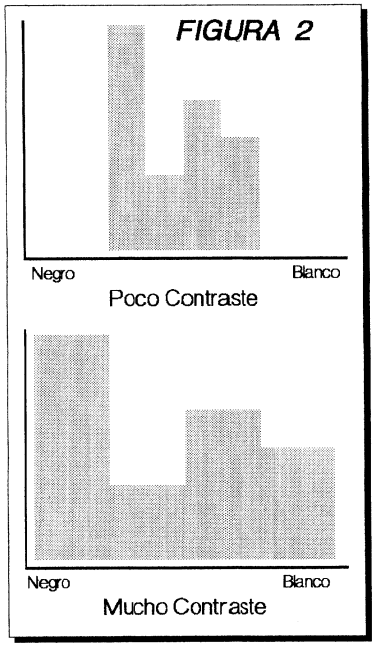
Brillo y Contraste

Esto nos da una idea de como modificar el brillo y contraste de las imágenes. Si queremos aumentar el brillo de una imagen solo tendremos que aumentar las intensidades de los pixels en la pantalla, de esta forma estamos desplazando el histograma hacia las cercanías del blanco y por tanto aumentando el brillo total de la imagen. A la vista de este resultado nos planteamos inmediatamente el siguiente problema. ¿Qué pasa cuando lleguemos al final de nuestro rango dinámico de colores (todos los que podemos representar?) Bien, el resultado final, independientemente de como se realice el ajuste del brillo será que la imagen se convertirá en un rectángulo blanco si aumentamos demasiado el brillo, la imagen se satura (esto se puede comprobar aumentando el brillo de nuestra televisión).

Si aumentamos demasiado la intensidad de gris de cada pixel de nuestra imagen llegará un momento en el que nuestro histograma llegue al tope máximo y empecemos a perder infor-

mación. Los valores que representan el blanco ya no pueden aumentar más. Una forma de subsanar esta deficiencia es que en lugar de desplazar el histograma desplazamos el rango dinámico en el que se define éste, es decir, redefinimos la paleta para que los colores que no se utilizarían en la parte inferior se aprovechasen para representar los colores que se perderían por la parte superior. Con este método tendríamos que realizar las cuentas sobre los valores que definen los tonos de gris sobre la paleta, puesto que los índices ahora carecerían de significado (el índice 0 no va a representar el blanco, sino un gris).

Para modificar el contraste la cosa ya cambia, puesto que debemos de expandir o comprimir el histograma. La forma más sencilla para realizar esta operación es mediante una tabla *look-up*. Los índices de la tabla representan el color de entrada y el valor de cada elemento de esa tabla el valor en el que se transforma, es análogo al método que utilizamos para la ordenación de los tonos de gris en el artículo anterior. Para comprender mejor como realizar esta operación estudiemos el problema a la luz de una gráficas (figura 3).



En estas gráficas (denominadas funciones de transferencia, puesto que relacionan los parámetros de entrada con los de salida) el eje de abcisas (eje x) representa los valores de los pixels de la imagen de entrada y el eje de ordenadas (eje y) el valor que le asigna la función de transferencia. En el *ejemplo 1* tenemos una imagen muy oscura y con poco contraste, todos los valores agrupados en torno al cero, se trataría de una imagen en la que apenas podríamos ver un borrón negro. La función de transferencia asigna a cada entrada de la imagen inicial una salida que cubre todo el rango dinámico (posibles valores que puede tomar la imagen) disponible con lo que obtenemos una imagen de alto contraste en la que ya será posible distinguir algo. Este problema aparece muy a menudo con sistemas de adquisición de imágenes en los que se obtienen imágenes demasiado oscuras, o claras (o grises), en la que todos lo valores están muy próximos y en la que es imposible distinguir nada. El ejemplo 2 representa el caso contrario en el que a una imagen se le reduce el contraste comprimiendo sus valores.

Aquí vuelve a aparecer el mismo problema descrito para el ajuste de brillo debido a la resolución finita de los computadores (en nuestro caso 8 bits, 256 valores). Cuando comprimimos el histograma si hay entradas en todos habrá valores que coincidan entre dos de los posibles representados, es decir, si vamos a pasar de un

rango dinámico de 0 a 255 a uno de 50 a 100, tenemos que meter los 256 valores en solo 49 (puesto que solo podemos trabajar con números enteros) con lo que varios pixels tomarán el mismo valor y perderemos información.

Una solución a estos problemas consiste en una ampliación del rango dinámico mediante una redefinición de la paleta, es decir, nosotros disponemos de 256 valores en los que inicialmente el 0 es negro y el 255 es blanco, estos 256 valores los convertimos en 49 (de 50 a 100), con lo que estaremos utilizando solamente 49 valores de los 256 de los que disponemos, así pues, una solución consistiría en una redefinición de la paleta de colores, de tal forma que el 0 ahora pase a representar el anterior 50 y el 255 el anterior 100, de esta forma dispondríamos de 255 valores entre 50 y 100. Esta solución está limitada por la capacidad gráfica de nuestro sistema, por ejemplo, un ordenador que permita definir una paleta de grises de 10 bits, si nosotros sólo tomamos valores para nuestra imagen de 8 bits (0-255) las entradas de la paleta contendrán múltiplos de 4 (tomamos valores equiespaciados, y como sobran dos bits...). La entrada cero será el cero, la entrada 1 será el 4, la entrada 2 será el 8,... puesto que pretendemos cubrir el mayor rango dinámico de colores (de negro a blanco). Así la entrada 50 contendrá el 200 y la entrada 100 contendrá el 400, con lo que entre 50 y 100 ahora podemos redefinir 200 tonos de gris más. Tomándolos equiespaciados la entrada 0 de la paleta será el 400, la entrada 1 será el 200+200/255, aproximado a un entero, y para el 255 tendríamos el 200+255*200/255=400 con lo que tenemos 255 colores entre el gris 200 y el gris 400 (ojo, estos valores son con una paleta de 10 bits, en ella el 0 representa el negro y 1023 representa el blanco, tenemos más resolución).

Las funciones de transferencia nos permiten modificar el histograma a nuestro gusto y su implementación es inmediata mediante una matriz.

pixel de salida = Matriz[pixel de entrada]

Para terminar describiremos una serie de algoritmos comúnmente utilizados para realzar imágenes y que incorporan la mayoría de los paquetes de software dedicados a esta tarea. Comenzaremos por la técnica conocida como ecualización de imagen. Esta técnica, permite obtener una imagen de alto contraste, expandiendo el histograma de una forma uniforme a lo largo del rango dinámico de valores. Esta técnica resulta muy útil con imágenes con su histograma comprimido en un cierto rango de valores, los cuales por ejemplo corresponden al fondo de la imagen. Como ya hemos indicado anteriormente, esto supone que el fondo es prácticamente un borrón. El ecualizado de imagen permite en la mayoría de los casos obtener un resultado bastante bueno. En la bibliografía indicada se incluyen detallados análisis estadísticos de como se obtiene la función de transferencia con la cual obtenemos esta distribución uniforme.

Comenzamos calculando el histograma de la imagen. De la definición de histograma resulta inmediato codificar un programa que lea todos los pixels de la imagen e incremente la entrada de la matriz que el mismo indica.

Histograma[pixel_imagen]++;

A continuación determinamos la función de transferencia que no es más que la suma acumulada de los valores del histograma escalada por un

factor (255/Max_Suma) para que de esta forma el máximo valor de la suma sea 255, nuestro máximo valor de representación.

Una vez determinada la función de transferencia, o mejor dicho, la matriz de transferencia, solo tenemos que aplicarla a la imagen de la forma indicada anteriormente.

Otra de las opciones comunes en un paquete de software dedicado al procesado de imagen es la de negativo. Su implementación es inmediata, puesto que no tenemos más que convertir el blanco (255) en negro (0), el gris claro (254) en gis oscuro (1), y así sucesivamente. La forma de realizar esto es restando al valor máximo (255) el valor del pixel actual. De esta forma el 255 pasa a ser 0 y el 0, 255. Por lo tanto la operación a realizar es:

pixel_salida = 255−pixel_entrada;

Otra opción común es la conocida como *"thresholding"* palabra inglesa que significa algo así como umbralizar. Ya hemos hablado de ella cuando describíamos los métodos para obtener imágenes monocromas que pudiéramos representar en dispositivos de la misma condición (impresoras, pantallas de plasma, etc..). Consiste simplemente en marcar un umbral, un valor de gris. Los pixels cuyo valor esté por encima del umbral se convertirán en blanco (255), los que estén por debajo en negro (0).

Una última técnica, más bien se trata de un efecto curioso, que ya incorporan muchos video digitales. El mosaico. El mosaico consiste en una división de la imagen a modo de tablero de ajedrez y asignar a cada uno de los bloque así definidos un valor constante que en general será la media de intensidades de los pixels dentro de ese bloque. Cuanto mayor sea el bloque menor resolución espacial tendrá la imagen y por tanto menor calidad.

Si en lugar de pintar todo el bloque con el valor medio anteriormente citado, pintamos solamente un pixel lo que estaremos haciendo es una especie de zoom inverso, reduciendo la resolución de la imagen. Si realizamos la operación "inversa" (lo ponemos entre comillas por que sólo es inversa en relación a lo del zoom... bueno podríamos decir recíproca), esto es en lugar de tomar un bloque, calcular su inten-

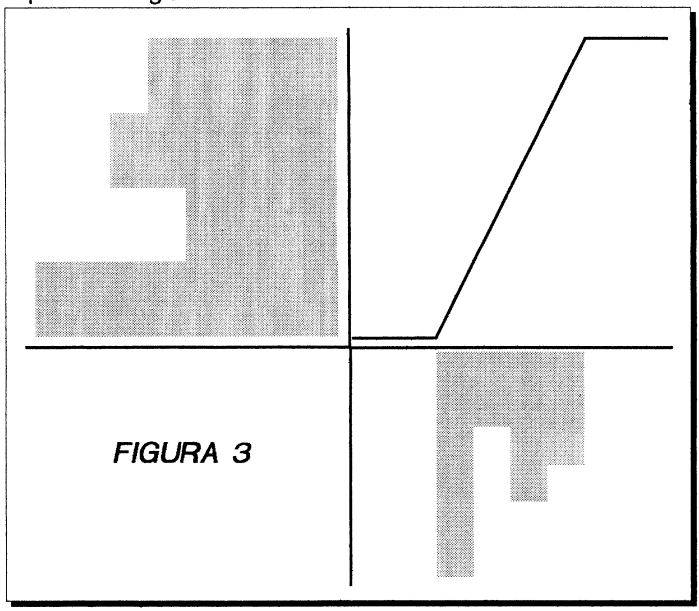


FIGURA 3

sidad media y asignar esta a todos los pixels en ese bloque, tomamos cada pixel de la imagen y pintamos un bloque de n*n pixel con ese color. De esta forma estamos realizando una especie de zoom (este tipo de ampliación lo hemos visto también en algunos videos digitales).

En próximos artículos hablaremos del problema del remuestreo de una imagen (resampling), siendo esta última la técnica más sencilla para llevarlo a cabo.

Nota de los Autores: En la mayoría de los libros sobre procesado de imagen, la parte de tratamiento de brillo y contraste es bastante escasa y las explicaciones son quizás demasiado ambiguas, a excepción, claro está, de la técnica del ecualizado. Por tanto la mayor parte de lo que se describe en este artículo sobre brillo y contraste es fruto de la experiencia personal de los autores con paquetes de software comercial y el propio software desarrollado por nosotros mismos.

¡¡ NECESITAMOS TU AYUDA !!

DATA BUS necesita tu ayuda para poder salir a la calle. Necesitamos tus artículos, sugerencias, comentarios y críticas.

Colabora con nosotros en la difusión y el mejor conocimiento de la informática. Si tienes algo que decirnos o deseas contar algo relacionado con nuestra afición, ¡¡ escribe YA!!.

Contamos con vosotros. Gracias Anticipadas.

Algoritmos de

clasificación (II)

Jesús Cea

En este número publicamos la segunda y última (probablemente) parte de la serie sobre algoritmos de clasificación alfabética, dedicado a los algoritmos de clasificación alfabética avanzados. Se caracterizan por tener una complejidad cuasilineal en vez de cuadrática.

En el número anterior de *DATA BUS* analizamos algunos de los algoritmos de clasificación alfabética más sencillos. Se caracterizaban, fundamentalmente, porque el tiempo de ordenación era proporcional al cuadrado del número de elementos a clasificar. A pesar de ello, cuando el número de datos es relativamente pequeño, estos sistemas pueden funcionar mejor que otros algoritmos teoricamente más eficientes, pero que necesitan trabajar con un número grande de elementos para poder sacar partido de su mayor potencia.

El umbral de decisión entre unos u otros algoritmos depende mucho de la codificación concreta, además. Como suelo decir a menudo, lo mejor es programar varios casos y estudiar cual se adapta mejor a nuestras necesidades.

Si en el número anterior estudiábamos algoritmos cuyo tiempo de ejecución era proporcional al cuadrado del número de elementos a ordenar, el objetivo de esta parte será el analizar el funcionamiento de unos cuantos algoritmos más evolucionados. Estos tendrán, como principal característica, un tiempo de clasificación proporcional a 'n*$\log_2$(n)', donde *'n'* es el número de datos con los que estamos trabajando. Como ya comenté en un párrafo anterior, estos algoritmos suelen ser bastante sofisticados y sólo se les saca partido cuando *'n'* es relativamente grande. En casos intermedios puede ser conveniente la utilización de algoritmos *"a caballo"* entre ambos extremos, tal como el *Shellsort* que estudiaremos a continuación:

SHELLSORT

En el número anterior dejé planteada una cuestión curiosa cuya respuesta no resulta evidente: ¿por qué la familia de métodos de la burbuja es tan popular? A juzgar por los tiempos (completamente verídicos y fiables, ojo) publicados en la primera parte de esta serie, son bastante inferiores al resto de los algoritmos vistos. ¿Cuál es, por tanto, su secreto?

En su día hubo quien me comentó, a título personal, que encontraba mucho más intuitivo el funcionamiento de la burbuja que el de selección o inserción. Personalmente no opino lo mismo, pero ello no afecta a nuestra pregunta.

¿Cuál es la razón de su *"éxito"*? La clave está en la tabla de tiempos de ejecución. Si leéis la primera línea encontraréis la clave: *"los tiempos de ejecución vienen dados en segundos y se han calculado para un conjunto de datos completamente desordenado"*. La cuestión es la siguiente: ¿las cosas serían diferentes si los datos no estuviesen desordenados del todo? Pues la verdad es que sí, como ya se comentó en el número anterior. No voy a repetir lo que dije en la primera parte, así que os recomiendo que la releáis.

Como resumen, notar que el comportamiento de los algoritmos de la burbuja es muy dependiente de los datos de entrada. Se comporta muy bien, por ejemplo, con matrices casi ordenadas.

¿Cómo mejorar los métodos de la burbuja para datos desordenados? Ya comenté en el número anterior que uno de los puntos flacos de la familia de la burbuja es que la mayor parte de los movimientos de datos son superfluos. Si pudiésemos "acercar" los datos a sus posiciones finales podríamos reducir de forma importante el número de intercambios. ¿Cómo? Pues muy fácil. Veamos todo ésto con más atención.

Resulta inmediato comprobar que los algoritmos de burbuja realizan intercambios entre elementos adyacentes. ¿Qué ocurriría si intercambiásemos datos más distantes? Normalmente se utilizan valores de "salto" según un polinomio de la forma *"a*n+1"*, con *'n'* variando desde cero. Obsérvese que cuando *'n'* vale cero el algoritmo degenera a burbuja. Sin embargo los intercambios más alejados ya han sido efectuados... El valor de *'a'* es algo arbitrario y en la tabla de tiempos del final podéis ver los resultados para distintos valores.

SHELLSORT

```
#define FALSE 0
#define TRUE !FALSE

void shellsort(int datos[])
{
int cont,pos1,pos2,paso;
int buff,ok;

  for(cont=MAXDATOS/3;cont>=0;cont--) {
    paso=3*cont+1;/* Paso del algoritmo */
    do {
      ok=TRUE;
      for(pos1=1,pos2=paso+1;pos2<=MAXDATOS;
pos1=pos2,pos2+=paso) {
        if(datos[pos1]>datos[pos2]) {
        buff=datos[pos1];
        datos[pos1]=datos[pos2];
        datos[pos2]=buff;
        ok=FALSE;
        }
      }
    } while(ok==FALSE);
  }
}
```

He leído en algún sitio que *"3*n+1"* es óptimo, lo que parece confirmarse en la tabla. En mi opinión los valores de salto óptimos son los números primos decrementados en uno pero como son bastante más difíciles de generar que un polinomio... Obsérvese, así mismo, que los tiempos son proporcionales al cuadrado de elementos, por lo que este sistema no resulta eficiente cuando *'n'* es grande.

ALGORITMOS CUASILINEALES

Hasta ahora hemos analizado algoritmos de clasificación de complejidad cuadrática, es decir, que el tiempo de proceso es proporcional al cuadrado de elementos a ordenar. Cuando se trabaja con pocos datos pueden ser bastante útiles por su sencillez programativa y conceptual, pero cuando nos

QUICKSORT 1

```
#define FALSE 0
#define TRUE !FALSE

void quicksort(int datos[],int datos2[],int inicio,int
final) {
int cont,pos1,pos2;
int buff,valor;

  if(final<=inicio) return;
  pos1=inicio;
  pos2=final;
  valor=datos[pos2];
  for(cont=inicio;cont<final;cont++) {
    buff=datos[cont];
    if(buff<valor) {
      datos2[pos1++]=buff;
    } else {
      datos2[pos2--]=buff;
    }
  }
  datos2[pos1]=valor;
  quicksort(datos2,datos,inicio,pos1-1);
  quicksort(datos2,datos,pos2+1,final);
  for(cont=inicio;cont<=final;cont++)
  datos[cont]=datos2[cont];
}
```

enfrentamos a un conjunto de valores "realista" vemos que las cosas no son tan alagüeñas. Por ejemplo, supongamos un algoritmo que tarda una milésima de segundo en ordenar 1000 elementos. Así, a simple vista, parecería un sistema bastante "decente", ¿no?. Pongámoslo ahora a clasificar un millón de elementos; tarda casi 17 minutos... y eso sin considerar la programación adicional necesaria para poder manejar tantos datos (por ejemplo, trabajando en disco).

Ante este problema hay tres soluciones fundamentales: la primera y más obvia (y la que realiza mucha gente, para desgracia de la informática) consiste en comprarse un ordenador "más grande". La segunda opción, también bastante obvia, es reprogramar el algoritmo empleado utilizando un lenguaje más cercano a la máquina, como el ensamblador. No obstante ninguna de estas dos opciones resulta práctica, ya que sólo estamos retrasando la aparición del problema. Supongamos, por ejemplo, que conseguimos acelerar 100 veces el algoritmo anterior mediante la compra de un nuevo equipo y la utilización del ensamblador. Bastaría tener que manejar diez millones de datos para estar en las mismas...

Aún nos queda una tercera posibilidad: cambiar de algoritmo. Afortunadamente existen algoritmos mucho más eficientes para la clasificación alfabética, algoritmos de complejidad "cuasilineal". Esto quiere decir que los tiempos son proporcionales al número de elementos, y no a su cuadrado. Obviamente los tiempos también son crecientes, pero la tasa de crecimiento es muchísimo menor. En realidad los tiempos son proporcionales a *"n*log₂(n)"*, pero a medida que *'n'* va creciendo el término asociado al logaritmo tiene menor importancia, de ahí el nombre.

Estudiaremos los dos sistemas más conocidos, denominados *Quicksort* (clasificación rápida) y *Mergesort* (clasificación por mezcla). Ambos se basan en la técnica de *"divide y vencerás"* y su formulación más intuitiva es la recursiva (también se pueden programar de forma iterativa y resultan ser más rápidos, pero se desvirtúa la idea original y resultan más difíciles de entender y de seguir).

La idea fundamental en torno a la cual giran estos algoritmos es la siguiente: si clasificar *'n'* elementos nos supone *'n²'* unidades de tiempo (prescindimos de las constantes de proporcionalidad en aras de la claridad), clasificar dos conjuntos de datos de *'n/2'* elementos cada uno nos llevaría *¡¡¡la mitad del tiempo!!!*. Si ahora dividimos cada conjunto de *'n/2'* elementos

en dos de *'n/4'* tardaríamos la cuarta parte, etc. En realidad las cosas no son tan fáciles, ya que estamos ordenando conjuntos independientes y luego hay que mezclarlos, lo cual también nos lleva tiempo. Y es precisamente la forma de mezclar los conjuntos la mayor diferencia entre el *Quicksort* y el *Mergesort*, como veremos.

Por cierto, en realidad los algoritmos no dividen el conjunto de datos original en dos, luego en cuatro, etc., sino que proceden exactamente al revés: parten de conjuntos de un elemento que mezclan para formar conjuntos de dos elementos, luego de cuatro, de ocho, etc. (es lo más normal cuando se utiliza una programación recursiva). Hay *'$log_2(n)$'* pasos y en cada uno hay *'2^s'* conjuntos de *'$2^{log(n)-s}$'* elementos, donde *'s'* es el número de paso. Así mismo se pueden mezclar dos listas ordenadas para formar otra también ordenada en un tiempo proporcional a la suma del número de elementos (si ambas listas tienen los mismos), por lo que mezclar dos listas de la etapa *'s'* supone *'$2*2^{log(n)-s}$'*, y como hay que hacer *'2^{s-1}'* mezclas, el tiempo total empleado en cada etapa resulta ser proporcional a *'$2^{log(n)}$'* o, lo que es lo mismo, *'n'*. Y como hay que hacer *'log(n)'* etapas, de ahí sale el *'n*log(n)'*.

Espero que hayáis entendido el "rollo" anterior. Desde luego no es necesario conocer el funcionamiento de los listados para poder utilizarlos, pero recordad que lo que pretendo es que comprendáis como hacen lo que hacen, no que copiéis las rutinas directamente en vuestros programas...

QUICKSORT

Pero dejémonos de teoría y vayamos al grano. El primer algoritmo que vamos a ver se llama *Quicksort* (clasificación rápida). Curiosamente no es el algoritmo de clasificación más rápido que se conoce (supongo que cuando se inventó sí...), pero como ese es el nombre habitual que se le da seguiremos la nomenclatura establecida.

Antes de pasar a las rutinas en sí voy a explicar someramente su funcionamiento, ya que resulta algo complicado de ver estudiando sólo los listados. Suponed un conjunto de números aleatorios, que serán los valores a clasificar. Tomemos ahora uno al azar, no importa cual, y extraigámoslo del conjunto. Tomando ese valor como referencia vamos leyendo los números que quedan uno a uno y generamos dos conjuntos nuevos, uno formado por los números mayores y otro constituido por los números menores. Al final tenemos dos conjuntos de datos, uno de números mayores y otro de números menores. Clasifiquemos ambos conjuntos por cualquier método.

QUICKSORT 2

```
void quicksort(int datos[],int inicio,int final)
{
int pos1,pos2;
int buff,valor;

  if(final<=inicio) return;
  pos1=inicio;
  pos2=final;
  valor=datos[pos2--]
  for(;pos1<=pos2;) {
    buff=datos[pos1];
    if(buff<valor) {
      pos1++;
    } else {
      datos[pos1]=datos[pos2];
      datos[pos2--]=buff;
    }
  }
  buff=datos[pos1];
  datos[pos1]=valor;
  datos[final]=buff;
  quicksort(datos,inicio,pos1-1)
  quicksort(datos,pos2+2,final);
}
```

Ahora basta con poner uno a continuación del otro, insertando en medio el valor que habíamos sacado al principio, para tener el conjunto inicial completamente ordenado.

A la hora de clasificar cada uno de los dos conjuntos generados podemos repetir el proceso: tomamos un valor de forma aleatoria y luego dividimos el conjunto inicial en dos, uno formado por los valores mayores y otro por los menores, ordenamos cada uno por separado y luego los fusionamos de la forma descrita. El proceso de división continúa hasta llegar a conjuntos de tamaño cero o uno. En la figura 1 podéis seguir el proceso mediante un ejemplo.

El primer listado *Quicksort* hace precisamente ésto. En vez de tomar un elemento al azar coge siempre el último. Si la lista está desordenada los resultados son los mismos. Lo he programado así porque resulta más

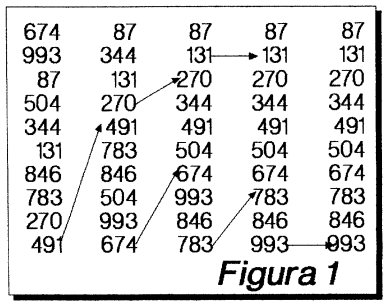


Figura 1

sencillo y para evidenciar el problema que tiene el algoritmo *Quicksort*, más adelante. Se puede ver que se utilizan dos matrices para facilitar la creación de los dos conjuntos en cada etapa.

El hecho de utilizar dos matrices simplifica la programación, pero a costa de una mayor cantidad de memoria y a una mayor lentitud. Esto es así porque cada vez que hay que dividir un conjunto, se copia cada uno de sus elementos a uno de los dos nuevos conjuntos, luego esos conjuntos se procesan (más operaciones) y una vez que están ordenados se vuelven a copiar sobre el conjunto original. Son muchos movimientos, ¿no?

Como demuestra el siguiente listado es posible hacer las operaciones *"in-place"* (como la F.F.T....) utilizando sólo una matriz, ahorrándonos memoria y tiempo. Las cosas ya no se ven tan diáfanas y evidentes, claro, pero es el precio que hay que pagar. Resulta interesante estudiar el uso de los índices *'pos1'* y *'pos2'*, uno señalando al conjunto de datos menores y otro al conjunto de datos mayores (o iguales). No voy a explicar cómo es que funciona, basta que lo analicéis con cuidado. Observando la tabla de tiempos publicada al final se ve que se gana bastante respecto al listado anterior. Ello es debido, fundamentalmente, al hecho de ahorrarnos las copias de una matriz a la otra.

QUICKSORT 3

```
void quicksort(int datos[],long inicio,long final)
{
long pos1,pos2;
int buff,valor;
  if(final<=inicio) return;
  pos1=inicio;
  pos2=final;
  valor=datos[pos2--];
  buff=datos[pos1];
  for(;pos1<=pos2;) {
    if(buff<valor) {
      buff=datos[++pos1];
    } else {
      datos[pos1]=datos[pos2];
      datos[pos2--]=buff;
      buff=datos[pos1];
    }
  }
  buff=datos[pos1];
  datos[pos1]=valor;
  datos[final]=buff;
  quicksort(datos,inicio,pos1-1);
  quicksort(datos,pos2+2,final);
}
```

El siguiente listado funciona exactamente igual que el anterior pero tiene algunas modificaciones en la codificación. En realidad no debería publicarlo, ya que no incorpora ninguna novedad y aún encima es más lento... Esto puede cambiar según la arquitectura en la que se programe, claro...

Bueno, ha llegado el momento de realizar un pequeño repaso. En los listados comentados hasta ahora se toma siempre el último elemento como elemento *"clave"* a la hora de dividir el conjunto en dos. Podríamos haber tomado el primero, el del medio, el que está a 3/5 del final o, incluso, uno al azar. Da exactamente lo mismo. ¿O no? Pues no, al menos no exactamente. Las cosas –como suele ocurrir– no son tan sencillas. Observad la figura 2, por ejemplo. En ella se muestra el funcionamiento de las rutinas vistas cuando actúan sobre una lista ya ordenada. Un desastre. En vez de generar dos conjuntos con un número de elementos similar se generan dos conjuntos, pero uno de tamaño *'n–1'* y otro de tamaño cero (donde *'n'* es el número de elementos de la lista del nivel actual, no el número de elementos en total).

Entre otras cosas ello ocasiona que la profuncidad recursiva que se alcanza sea *'n'*; por ejemplo, si clasificamos un millón de elementos completamente desordenados podemos esperar una profundidad en torno a

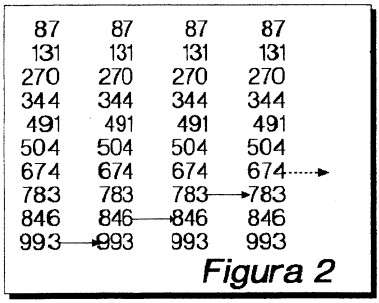


Figura 2

```
void quicksort(int datos[],int inicio,int final)
{
int pos1,pos2;
int buff,valor;
int temporal;

  if(final<=inicio) return;
  pos1=inicio;
  pos2=final;
  temporal=((unsigned int)pos1+pos2)>>1;
  valor=datos[temporal];
  datos[temporal]=datos[pos2];
  datos[pos2--]=valor;
  for(;pos1<=pos2;) {
    buff=datos[pos1];
    if(buff<valor) {
      pos1++;
    } else {
      datos[pos1]=datos[pos2];
      datos[pos2--]=buff;
    }
  }
  buff=datos[pos1];
  datos[pos1]=valor;
  datos[final]=buff;
  quicksort(datos,inicio,pos1-1);
  quicksort(datos,pos2+2,final);
}
```

QUICKSORT 4

20, pero si la lista estaba ya ordenada inicialmente llegamos a una profundidad de un millón y eso, amigos, es algo que ningún ordenador (o casi) es capaz de manejar... Además del problema de la profundidad de la recursión también nos enfrentamos a unos tiempos de cálculo mucho mayores a los que sería de esperar. A fin de cuentas no debemos olvidar que estamos echando por tierra los fundamentos mismos del algoritmo...

¿Cuál es la solución? Bueno, podíamos tomar un elemento al azar en cada paso. Otro sistema más simple consiste en tomar siempre el elemento central. Si la lista estaba desordenada da exactamente igual, y si ya estaba bien ordenada cada paso dividirá correctamente las listas en conjuntos de longitudes aproximada-

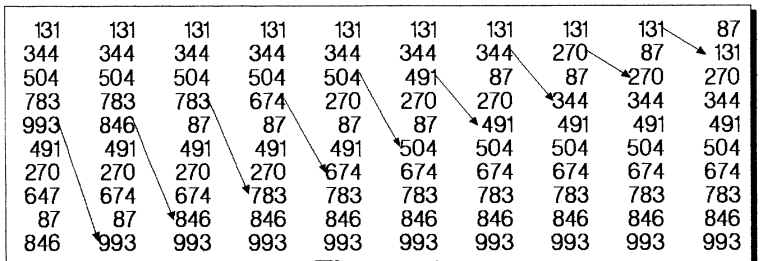


Figura 4

mente iguales, como queríamos. Eso es, precisamente, lo que hace el último listado *Quicksort* que publicamos. El funcionamiento es idéntico a los anteriores salvo en el hecho de que ahora tomamos el elemento central.

Es de destacar que resulta un poco más rápido incluso sobre listas desordenadas. Probablemente sea debido a que tan sólo hay 32768 valores diferentes para cada elemento (debido al *RND* utilizado) y cuando estamos trabajando con un millón de elementos es de esperar que cada valor se repita unas 30 o 31 veces, y este sistema es algo más regular. Pero la verdadera diferencia hubiera estado clara si se pone a trabajar sobre una lista ya ordenada. Si alguien hace la prueba con alguno de los listados anteriores (*Quicksort 1, Quicksort 2 o Quicksort 3*) le recomiendo que utilice pocos elementos (unos pocos cientos), y que compile el programa con un tamaño de pila lo más grande posible y con la opción de comprobación de rebosamiento de la misma activada...

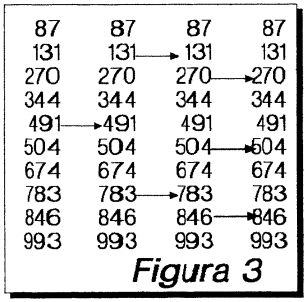


Figura 3

De todas formas no creáis que el listado *Quicksort 4* es la panacea. En la figura 4 podéis ver una lista que le provoca el mismo problema. Se trata de un fallo del algoritmo en sí, y aunque funcione correctamente el 99.999999% del tiempo, el hecho de que haya un solo caso en el que los tiempos se disparan (y eso suponiendo que la pila no rebose antes) hace que deje de ser un algoritmo perfecto para pasar a ser "casi"...

De todas formas utilizando el *Quicksort* 4 podemos estar razonablemente seguros de que la rutina va a portarse bien en todos los casos prácticos, así que no vale la pena complicarse más la existencia. Claro que siempre es posible introducir comprobaciones sobre el tamaño del anidamiento en cada instante, por ejemplo...

MERGESORT

Veamos nuestro último algoritmo. Se llama *Mergesort*, o clasificación por mezcla de listas. La idea básica es incluso más sencilla que en el caso del *Quicksort*: dividir cada conjunto en dos, clasificarlos por separado y luego fusionarlos. Además tiene otra ventaja interesante, y es que carece de "casos raros", no como le ocurre al *Quicksort*. Es más, puede decirse que

el tiempo de clasificación es basicamente *INDEPENDIENTE* de la configuración inicial de la lista. Es decir, tarda el mismo tiempo (aproximadamente) en ordenar una lista completamente aleatoria que otra ya clasificada en orden creciente o decreciente. A primera vista ésto pudiera parecer un problema (falta de eficiencia dirían algunos...), pero hay aplicaciones en las que interesa tener una cota para el tiempo o tener una estimación más o menos exacta del tiempo que se necesita. Esa propiedad ya la poseían algunos algoritmos vistos en el número anterior.

El algoritmo funciona de la forma siguiente. En cada etapa toma el conjunto que se le da y lo divide en dos conjuntos del mismo tamaño. Una vez que ambos conjuntos estén ordenados hay que mezclarlos, para lo cual se van comparando elemento a elemento y pasandose a otra lista. Hay que tener cuidado cuando uno de los conjuntos se quede sin elementos, tal y como se ve en el listado. En la figura 5 puede verse en acción.

Hay un problema: hacen falta dos listas, ya que las operaciones no pueden realizarse *'in-place'*, con el consiguiente consumo adicional de memoria. Obsérvese, no obstante, que ello no supone un enlentecimiento del algoritmo tal y como ocurría en el caso del *Quicksort*. Esto es así porque ese es, precisamente, la idea fundamental del algoritmo... Naturalmente es posible modificar la rutina para que se realicen los cálculos sobre la matriz original, ahorrándonos así mucha memoria. El listado publicado bajo el epígrafe *Mergesort 2* es un ejemplo de ello. Sin embargo los tiempos... bueno.. digamos que no son comparables...

El desarrollo de una rutina *Mergesort* eficiente (comparada con *Mergesort 1*) con una sola matriz queda como ejercicio para el lector...

```
void mergesort(int datos[],int datos2[],int inicio,int final)
{
int medio;
int pos1,pos2;
int contador;

  if(inicio>=final) return;
  medio=((unsigned int)inicio+final)>>1;
  mergesort(datos2,datos,inicio,medio);
  mergesort(datos2,datos,medio+1,final);
  pos1=inicio;
  pos2=medio+1;
  for(contador=inicio;contador<=final;contador++) {
    if(datos2[pos1]<datos2[pos2]) {
      datos[contador]=datos2[pos1++];
      if(pos1>medio) {
        for(contador++;contador<=final;contador++) {
        datos[contador]=datos2[pos2++];
        }
      }
    } else {
      datos[contador]=datos2[pos2++];
      if(pos2>final) {
        for(contador++;contador<=final;contador++) {
        datos[contador]=datos2[pos1++];
        }
      }
    }
  }
}
```

MERGESORT 1

Por cierto, dado el "determinismo" de este sistema se presta muy bien a la implementación mediante *"software cableado"*. La técnica a emplear está fuera de los objetivos de este artículo...

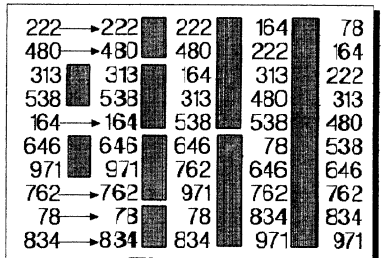


Figura 5

SOBRE LOS LISTADOS

El objetivo de los listados no es que los utilicéis directamente en vuestros programas sino el brindaros la oportunidad de que veáis y estudiéis la forma de implementar los sistemas comentados. Los "expertos" en *'C'* se habrán echado las manos a la cabeza al no haber observado ninguna construcción típica del lenguaje (si, ya sé que el Pascal -¡Puag!- también tiene punteros, pero no son lo mismo ya que, en el colmo de las restricciones, ni siquiera pueden señalar a los elementos de un array...). Las razones de no haber utilizado las "idiosincrasias" propias del *'C'* han sido puramente prácticas. Por un lado no tiene sentido "criptar" los listados (aunque con ello los programas fueran más eficientes) cuando lo que se pretende es, precisamente, que los lectores los comprendan. Por otra parte no todo el mundo conoce el *'C'* a la perfección y los programas se han codificado teniendo en mente la adaptación a otros lenguajes quizás más "vulgares" pero, sin duda, mucho más difundidos (léase Basic y Pascal).

Hablando de lenguajes, los ejemplos se han escrito para *"Pure C"*, sin duda el mejor compilador de *'C'* para Atari. Es compatible (a nivel de código fuente) con el *"Turbo C"*, así que los lectores que sean usuarios de PC no tendrán que modificar ni una coma. En realidad se trata de *A.N.S.I. C, por lo que* cualquier compilador moderno debería "tragar" los ejemplos sin ningún problema.

Para aquellos que no dominen el *'C'* paso a comentar, muy someramente, algunas peculiaridades de los listados. El *"void"* que aparece al principio sirve para indicar al compilador que la función no retorna ningún valor (se trata -en realidad- de un procedimiento). Los compiladores viejos (K&R) podrían no aceptarlo. *MAXDATOS* es el número de elementos a clasificar. En buena lógica debería ser un parámetro de la función, pero lo hice así por razones que no vienen al caso.

La admiración *(!)* conlleva la negación de lo que le sigue. En cuanto a los *"++"* y *"--"*, son los modos de autoincremento y autodecremento, construcciones que yo considero muy elegantes. Los *"#define"* sirven para declarar constantes. Observad la forma de definir *"TRUE"*. Impresionante, ¿eh? Bueno, creo que entre lo que he dicho, lo que sepáis sobre cualquier otro lenguaje, el sentido común y la indentación podréis desgranar el funcionamiento de las rutinas. Si hay alguna duda, consultad con vuestro profesor de informática o conmigo mismo...

MERGESORT 2

```
void mergesort(int datos[],int inicio,int final)
{
int medio;
int pos1,pos2;
int contador,contador2;
int buffer;

  if(inicio>=final) return;
  medio=((unsigned int)inicio+final)>>1;
  mergesort(datos,inicio,medio);
  mergesort(datos,medio+1,final);
  pos1=inicio;
  pos2=medio+1;
  for(contador=inicio;contador<=final;contador++) {
    if(datos[pos1++]<datos[pos2]) {
      if(pos1>medio) contador=final;
    } else {
      buffer=datos[pos2++];
      for(contador2=++medio;contador2>=pos1;
contador2--) {
        datos[contador2]=datos[contador2-1];
      }
      datos[contador]=buffer;
      if(pos2>final) contador=final;
    }
  }
}
```

TABLA DE TIEMPOS DE EJECUCION

Los tiempos de ejecución vienen dados en segundos y se han calculado para un conjunto de datos completamente desordenado. Los programas que se han utilizado para el cronometraje son los mismos que acompañan al artículo. Los tiempos son meramente orientativos y dependerán, obviamente, del compilador y la máquina. Los números en negrita que encabezan cada columna indican el número de elementos a procesar. Estos tiempos no pueden compararse directamente con los publicados en el número anterior, ya que se ha utilizado otra máquina y otro compilador (ver el texto).

Los algoritmos especificados de forma polinómica corresponden a variantes del algoritmo *Shellsort*, para diferentes polinomios de generación de "paso".

Se observa que los tiempos crecen, aproximadamente, de forma lineal con el número de elememtos, por lo que estos algoritmos se denominan *"cuasilineales"*. Las únicas excepciones son el *Shellsort* y el *Mergesort 2* Ello resulta comprensible, ya que el primero es una variante de los métodos de burbuja analizados en el número anterior, y el segundo malgasta una considerable potencia de cálculo a cambio de reducir la ocupación de memoria a la mitad (ver texto y listados).

Algoritmo	***1000***	***2000***	***5000***	***10000***	***20000***	***32000***
2*n+1	2.545	9.755	63.395	—	—	—
3*n+1	2.476	9.58	62.62	—	—	—
5*n+1	2.575	10.195	63.545	—	—	—
10*n+1	2.56	10.175	63.105	—	—	—
11*n+1	2.565	10.2	63.5	—	—	—
Quicksort 1	0.065	0.14	0.4	0.855	1.84	3.215
Quicksort 2	0.04	0.085	0.24	0.545	1.115	1.9
Quicksort 3	0.04	0.085	0.24	0.545	1.125	1.915
Quicksort 4	0.04	0.09	0.23	0.505	1.095	1.885
Mergesort 1	0.05	0.1	0.28	0.59	1.255	2.055
Mergesort 2	0.48	1.825	11.305	44.45	—	—

Aquellos lectores que detecten una *"discontinuidad"* en los tiempos al pasar de 32.000 a 100.000 elementos encontrarán la explicación en el texto.

Algoritmo	***100000***	***500000***	***1000000***
2*n+1	—	—	—
3*n+1	—	—	—
5*n+1	—	—	—
10*n+1	—	—	—
11*n+1	—	—	—
Quicksort 1	11.13	72.43	—
Quicksort 2	6.585	46.55	114
Quicksort 3	6.63	46.865	115.32
Quicksort 4	6.285	43.895	113.94
Mergesort 1	7.03	38.495	—
Mergesort 2	—	—	—

LOS TIEMPOS

Como se advierte en la tabla, los tiempos proporcionados son válidos sólo a la hora de clasificar una tabla completamente desordenada, salvo en el caso del algoritmo *Mergesort* (sus tiempos son aproximadamente constantes, como se comentó anteriormente). Por cierto, los tiempos no pueden compararse directamente con los publicados en el artículo anterior, ya que los programas han sido ejecutados en máquinas distintas (los anteriores en un Atari STE y éstos en un Falcon030), y utilizando también un compilador diferente.

Quizás algún lector avispado note que hay como una discontinuidad de en los tiempos al pasar de 32.000 a 100.000 elementos. Ello es debido a que en ese caso es necesario utilizar contadores, etc., de 32 bits en vez de 16. Indexando las tablas mediante punteros (una técnica típica y casi exclusiva del *'C'*, por eso no se ha empleado) se hubiera evitado ese "salto" (en 68000 se trabaja con memoria lineal, para el que no lo sepa), además de reducir los tiempos a sólo un tercio, aproximadamente...

CONCLUSION

Con este artículo se completa esta serie sobre algoritmos de clasificación alfabética. Naturalmente hay otros sistemas, algunos mejores y otros peores que los descritos. No obstante esta serie sólo pretende ofrecer a los lectores una visión general y no una lista exhaustiva de todo lo que hay por el mundo adelante. No debemos olvidar que muchos algoritmos están destinados a aplicaciones concretas o pretenden solucionar problemas muy específicos (por ejemplo, el hecho de que parte de los datos estén en disco). La imaginación que cada uno aplique en cada situación en particular también es importante.

Por ejemplo, sea una base de datos de nombres (un listín telefónico). Podemos utilizar *Quicksort* sobre ella, pero también podemos dividir la lista en 27 listas distintas más pequeñas, según la inicial del primer apellido, por ejemplo. Aplicando una técnica tan simple se reducirían tanto el tiempo de búsqueda como el de clasificación (en vez de *'n*log(n)'* sería *'n*log(n/s)'*, donde *'s'* es el número de conjuntos (1.31 más rápido en este caso). Si el algoritmo empleado fuera de complejidad cuadrática la ganancia sería mucho más espectacular (2700%). En la práctica no sería tanto, ya que no todas las listas medirían lo mismo; las listas de la *'a'* o la *'e'* serían mucho más grandes que las listas de la *'z'*, la *'w'* o la *'x'*. Esto se podría solucionar subdividiendo aún más las listas grandes teniendo, por ejemplo, una lista para la *'b'*, otra para la *'c'*, otra para los apellidos que empiezan por *'aa'* hasta *'an'*, otra para los que empiezan por *'ao'* hasta *'az'*, etc., intentando que todas las listas tengan un número de elementos similar. En fin, creo que cogéis la idea de lo que quiero decir con eso de la "imaginación".

Existen otros algoritmos interesantes, como el *Heapsort*, el *Radixsort*, el *Bucketsort*, etc., pero creo que con lo visto ya es suficiente. Es posible que escriba una tercera parte de esta serie, pero lo dudo. Lo que quería comunicar con ella ya lo he logrado... De todas formas, si hay alguien interesado en profundizar más en el tema sólo tiene que ponerse en contacto conmigo, como siempre...

¡¡ Noticias Frescas !!

Jose Manuel Suárez

Un número más, he aquí un pequeño resumen de las noticias más importantes o curiosas acaecidas en el mundillo de la informática en los últimos meses. Esperamos vuestras noticias en la dirección o el teléfono indicados en la página 2.

APPLE LICENCIA LA TECNOLOGIA NEWTON A TOSHIBA

Subrayando el creciente respaldo internacional a la plataforma Newton, Apple Computer ha incorporado a Toshiba Corporation al conjunto de empresas de todo el mundo que han adquirido licencias de la tecnología Newton de Apple. Toshiba tiene la intención de desarrollar una nueva línea de productos basados en tecnología Newton.

Toshiba Corporation era ya miembro fundador de la Asociación de la Industria Newton, la cual fue anunciada con ocasión de la primera Conferencia Internacional de Desarrollo Newton celebrada el pasado mes de Diciembre. Entre los participantes en la misma se encontraban importantes firmas de todo el mundo como Alcatel, ARM, BellSouth Mobile-Comn, British Telecom/Cellnet, Cirrus-Logic, Deutsche Telecom, GEC Plessey, LSI Logic, Kyushu Matsushita, Motorola, ParaGraph, Scriptel, Sharp, Siemens/ROLM, Telia, Traveling Software y US West. Todas estas compañías son licenciatarias de la tecnología Newton, OEMs, suministradores de componentes y aliados de Apple en el terreno del marketing relacionado con la tecnología Newton. Esta asociación es un fórum que promocionará el crecimiento e interoperabilidad de la plataforma Newton y los dispositivos asociados. Asimismo, la Asociación de la Industria Newton promoverá estándares para dispositivos Newton en las áreas de comunicaciones inalámbricas, soporte telefónico y ofimática.

La participación activa de Toshiba en la Asociación de la Industria Newton, al lado de importantes corporaciones vinculadas a muy diversos sectores industriales, contribuirá al creciente éxito y a la adopción generalizada de la revolucionaria plataforma Newton.

APPLE, IBM Y SCIENTIFIC-ATLANTA CREARAN UNA ARQUITECTURA ABIERTA PARA EL FUTURO MERCADO DE LA TV. INTERACTIVA

Apple Computer Inc, IBM Corporation y Scientific-Atlanta Inc han anunciado su intención de formar un equipo que combine recursos técnicos y de negocios, para trabajar conjuntamente en la creación de una arquitectura abierta para el futuro mercado de la televisión interactiva. Este equipo capitalizará el liderazgo y la fortaleza tecnológica de las tres compañías. El objetivo es construir interfaces para televisión interactiva que sean escalables, interoperables y abiertas, basadas

en las tecnologías actuales de estas tres importantes firmas. Este entorno operativo escalable será diseñado para su uso con terminales de comunicaciones domésticos (los denominados HCTs: Home Communications Terminals).

El propósito es desarrollar una arquitectura de sistemas que describa íntegramente un entorno de operación, la cual incluiría el modelo de aplicación *ScriptX* de Kaleida (la empresa conjunta de Apple e IBM), las interfaces de modelización de objetos denominadas *SOMobjects/DSOM* y *OpenDoc*, y la familia de microprocesadores *PowerPC*.

ScriptX es un lenguaje de alto nivel orientado a objetos y una tecnología de software de sistema concebido para crear títulos multimedia en una amplia diversidad de plataformas digitales: desde terminales de comunicación domésticos, hasta consolas de juegos interactivos y estaciones de trabajo de altas prestaciones.

"Este proyecto se asienta en nuestro trabajo en Apple y en Kaleida Labs. y encaja con nuestra visión a largo plazo de una arquitectura multimedia abierta y multiplataforma", ha dicho David Nagel, vicepresidente y director general de la división AppleSoft de Apple. *"Nos complace trabajar con líderes de la industria como Scientific-Atlanta e IBM para impulsar el desarrollo de la TV interactiva y de otros servicios multimedia interactivos"*.

"El denominado 'set-top box' (dispositivo que se coloca junto al televisor) es el mecanismo que incorporará a los suscriptores a la superautopista de la información", afirma James A. Cannavino, vicepresidente de estrategia y desarrollo de IBM. *"IBM, Scientific-Atlanta y Apple conjuntan una base única e incomparable de tecnologías y conocimientos, necesarios para desarrollar un set-top box potente y fácil de utilizar. Queremos combinar las mismas tecnologías clave que IBM está utilizando en los sistemas más avanzados, con el conocimiento de Scientific-Atlanta del mercado de los set-top. Nuestro objetivo es producir arquitecturas abiertas y escalables que estimulen el desarrollo de nuevas aplicaciones, y que se integren en las redes de televisión interactiva"*.

El consejero delegado de Scientific-Atlanta, James F. McDonald, añade: *"El esfuerzo conjunto de Apple, IBM y Scientific-Atlanta concentra un repertorio de recursos y conocimientos único para desarrollar un entorno de operación escalable y herramientas de autor para redes de banda ancha en el ámbito doméstico. En un entorno de arquitectura abierta, las empresas independientes podrán desarrollar múltiples e innovadoras aplicaciones a utilizar con los terminales de comunicaciones domésticos, lo cual ofrecerá a los clientes una amplia variedad de opciones de información y entretenimiento"*.

PREGUNTA

¿Cómo es posible comprobar la afirmación de algunos constructores de discos duros de que sus mecánicas tienen un tiempo medio de funcionamiento entre fallos (MTBF) de x horas?

Parece una cosa trivial, pero esta simple pregunta ha generado un entusiamo tal en *Internet* que las discusiones se prolongan todavía, después de meses y millones de pulsaciones de teclas.

Algunas de las repuestas han sido tan graciosas y (¿poco?) inteligentes como ésta:

"Puede que tenga una respuesta acerca de como determinar un MTBF de 500.000 horas. Compre 500.000 discos duros y enciéndalos todos a la vez.

Observe y anote cuantos de ellos dejaron de funcionar después de una hora.

Si unicamente uno de ellos no funciona, entonces tenemos un MTBF de 500.000 horas. Sin embargo, asegúrese de tener un amplio surtido de enchufes antes de probarlo en su propia casa..."

MAC VERSUS PC

Repasando la conferencia MAC/DOS Warz, que es la equivalente a "*Mac versus PC*" de Internet, pero en la red OneNet, he encontrado algunas joyas, pero especialmente un comentario que dice:

» *Hablar de ordenadores y sistemas operativos es como mantener el mismo diálogo que los tipos de Star Trek: La Nueva Generación con el Borg: "La resistencia es inútil. Seréis asimilados."*

› *[...] Bueno, pero, al final, ¿No fue el Borg eventualmente eliminado? Le hicieron confundirse y se perdió en la complejidad de todo el asunto."*

Muy buena la comparación...

¿COMPETENCIA AL POWERPC?

Parece que el reciente procesador *PowerPC, nacido de*l trabajo conjunto de IBM, Motorola y Apple, está causando auténtico terror entre sus competidores. Al menos eso parece tras la alianza entre Intel y Hewlett Packard para crear una nueva familia de procesadores compatibles, a nivel de código binario, con las arquitecturas Precision (PA) de H.P. y 80x86 de Intel.

Ciertamente hay razones para que estén preocupados. La relación precio-prestaciones de los *PowerPC* son excepcionales, y si a ello unimos que la asociación *PowerOpen* ha publicado los diseños completos de una familia de ordenadores basados en *PowerPC* y la puesta a punto de una máquina de referencia *PReP* (plataforma de referencia *PowerPC*) "lista para clonar", el futuro del *PowerPC* parece más que alagüeño. No olvidemos el efecto que ocasionó el que IBM permitiera "clonar" sus máquinas PC, hace unos 15 años...

Los expertos no parecen tomarse demasiado en serio este acuerdo entre Intel y H.P. La idea es buena, pero mantener la compatibilidad con la familia 80x86 -más que anticuada- parece un problema serio de diseño. Por lo pronto ni Intel ni Hewlett Packard han anunciado las técnicas que van a emplear, lo cual resulta prudente ya que las arquitecturas PA y 80x86 son radicalmente diferentes. ¿Cómo diseñar un procesador capaz de ejecutar (sin recompilación) de forma eficiente códigos tan distintos?

La idea que circula por ahí es que Intel permita a H.P. emular el código 8086 mediante una compilación "*al vuelo*" a código PA, realizándose todo ello de forma interna al microprocesador. IBM está utilizando una tecnología similar para el desarrollo del *PowerPC 615*, compatible 80x86.

De todos modos no parece que todo ésto les quite el sueño a los ingenieros de IBM, Motorola o Apple (ni a las compañías que las apoyan desde *PowerOpen*). Este hipotético microprocesador no estaría disponible hasta 1997 y para entonces los *PowerPC* ya estarían instalados en el mercado (incluidos sus modelos con capacidad de emulación 80x86)...

Y PARA ACABAR, UNO DE RUBIAS

...Una rubia y una morena paseando en el parque. La morena dice "*Oooh, mira el pobre pajarito muerto*". La rubia levanta la vista, mira al cielo y dice "*¿dónde?*"...

Grupo de Desarrollo Infomático

Desde el último número de *DATA BUS* han ocurrido muchas cosas, algunas buenas y otras no tanto. Entre las noticias buenas, sin duda alguna, se cuenta la creación del *Grupo de Desarrollo Infomático (G.D.I.)*.

El *Grupo de Desarrollo Informático* nació el pasado Noviembre en la *Universidad de Vigo*, más concretamente en la *Escuela Técnica Superior de Ingenieros de Telecomunicación*, como respuesta a la falta de iniciativas oficiales relacionadas con la informática y destinadas a los estudiantes.

Los objetivos del *G.D.I.* son muchos y muy variados: poner en contacto a los programadores entre sí, realización de proyectos en equipo, desarrollo de software en varias plataformas simultaneamente, realización de programas de apoyo al estudiante, cursillos, asesoramiento, facilitar el acceso a documentación técnica, investigación, etc. Para ello se recurre al concepto de *Grupo* o *Equipo*, siendo éste la unidad básica de funcionamiento del *G.D.I.* En la actualidad existen numerosos grupos de trabajo activos: Comunicaciones digitales, fractales, raytracing, compresión, procesado de imagen y sonido, criptación y autentificación, gráficos, compiladores, sistemas operativos, sonido, música, etc. A pesar de lo ambicioso de la idea y de lo irrealizable (o irreal, según se mire...) que parece, la verdad es que ya se están haciendo algunas cosas interesantes.

Quizás la más visible de ellas sea la revista *G.D.I.* La revista *G.D.I.* es una revista en disco ***"multiformato"***, lo cual quiere decir que funciona en *Amiga*, *Atari* y *PC compatibles* ***(¡¡el mismo disco!!)***. Ya está en la calle el número uno, compuesto por más de ochenta artículos, que impresos ocuparían más de 400 páginas en formato A4, y están ultimando ya el número dos (se espera que salga en Octubre). Naturalmente la revista es completamente gratuita y puede conseguirse en la red de la *Escuela Técnica Superior de Telecomunicaciones de la Universidad de Vigo*, en el *S.P.T.* (Servicio de Publicaciones de Teleco, de la misma Escuela) y en una amplia colección de localizaciones *FTP* repartidas alrededor de todo el mundo.

Podéis poneros en contacto con ellos a través de nuestra misma dirección (ver página dos), ya que algunos de nuestros socios desempeñan también cargos de responsabilidad en el *G.D.I.*, así como en la siguiente dirección de correo electrónico (para aquellos con acceso a Internet):

GDIØDTC.UVIGO.ES

(El símbolo Ø es la ARROBA)

Historia de la informática (IX)

Nacho Agulló

En este número veremos algunos de los esfuerzos norteamericanos en el campo de las computadoras, así como el nacimiento del primer ordenador electrónico.

Durante la Segunda Guerra Mundial, los esfuerzos de los contendientes en investigación se redoblaron. En las partes anteriores, hemos visto como los ingleses desarrollaban el *Colossus* y los alemanes el *Z3* y el *Z4*, siempre con miras a utilizarlos en el conficto. A continuación estudiaremos los trabajos de los norteamericanos, quienes pese a disfrutabar de una cómoda ventaja antes del estallido de la guerra, se veían superados por estas iniciativas de ingleses y alemanes.

Los físicos de Harvard

En 1939, la Universidad de Harvard llega a un acuerdo con IBM para desarrollar un ordenador llamado *Automatic Sequence Controlled Calculator (ASCC)*. El proyecto, comenzado dos años atrás por el físico Howard H. Aiken, pasa a conocerse como *Mark I*. Aiken, inspirándose en los trabajos de Babbage y Torres Quevedo, concibe una máquina electromecánica que emplea relés, ruedas dentadas y embragues electromagnéticos, entre otros elementos. El proyecto va a ser desarrollado por Aiken junto a científicos de su departamento e ingenieros de IBM. Esta empresa además aportará una subvención.

Una máquina monstruosa

Lauina, es presentada en Agosto de 1944. Su aspecto es impresionante: consta de 800 Km. de cable y 250.000 piezas, incluyendo 3.000 interruptores mecánicos y 7.000 relés; mide 15 metros de largo por 2.5 de alto, y pesa varias toneladas. Cuenta con elementos de entrada, memoria principal, unidad aritmética, unidad de control y elementos de salida, utilizando como entrada cintas y tarjetas perforadas.

En cuanto a su funcionamiento, es capaz de almacenar en su memoria (mediante ruedas dentadas) 72 números de 23 cifras decimales, más signo, que se introducían mediante las fichas perforadas. Las instrucciones son introducidas en las cintas perforadas, pudiendo programársele para ejecutar desde una simple suma hasta funciones trigonométricas y logarítmicas (sen x, e^x, etc.) a base de introducir las rutinas de cálculo de dichas funciones.

Potencia de cálculo

La efectividad del *Mark I* es inferior a los *Z3* y *Z4* de Zuse. Las operaciones de suma y resta se ejecutan

en tres décimas de segundo; la multiplicación de números de seis cifras lleva seis segundos, y la división, no menos de diez.

Globalmente, el *Mark I* no llega a la altura de las máquinas de Zuse; su velocidad en sensiblemente menor y además es más grande (lo que es una desventaja). Sin embargo, es un buen reflejo de la capacidad americana del momento, digna sucesora de los trabajos de Bush, Stibitz y demás. Posteriormente, Howard Aiken desarrollará una versión mejorada de la máquina, que se llamará *Harvard Mark II*, para la armada de los Estados Unidos. De características serán muy parecidas, no incorporando ninguna innovación notable. Este ordenador, ya en funcionamiento en 1945 (un año después de su predecesor) ser áempleado en cálculos balísticos y similares.

La introducción de la tecnología electrónica

La innovación que supuso la aparición del *ASCC* tenía también su cruz. En efecto, esta máquina lleva el cálculo mecánico más allá de lo que habían conseguido Bush y Stibitz; pero al mismo tiempo demuestra que la velocidad de los cálculos era dificilmente mejorable. Se hace preciso un salto cualitativo, un cambio de tecnología que permita otro orden de velocidades. Y esta nueva tecnología, como preconizaron Zuse y Atanasoff, va a ser la electrónica.

El primer ordenador electrónico

En 1940, la Escuela de Ingeniería Moore, de la Universidad de Pennsylvania, afronta la tarea de construir un ordenador cuyo nombre sería *ENIAC (Electrical Numerical Integrator And Calculator)*. Al frente del proyecto están John Eckert y John Mauchly, quienes se pondrán en contacto con nuestro viejo conocido Atanasoff. Así, enterados de los buenos resultados obtenidos con las nuevas técnicas que éste empleó en su *ABC*, pondrán en práctica la mayor parte de éstas en el *ENIAC*.

Como no podía ser de otra forma en plena guerra, el proyecto se ve vinculado al ejército a partir de 1943 y de esta forma, su primera función será... calcular trayectorias balísticas (para variar). Sin embargo, el *ENIAC* no se terminaría hasta terminada la guerra: el 15 de Febrero de 1946, concretamente, será el día de su puesta en funcionamiento.

Una máquina más monstruosa aún

El *ENIAC* se realiza utilizando nada menos que 18.000 válvulas electrónicas y 50.000 conmutadores, y ocupa todo el primer piso de la Escuela. Pesa 30 toneladas y consume de 150 Kilowatios; según la leyenda, las luces del barrio oeste de Filadelfia parpadean cada vez que el ordenador se enciende. Un pequeño ejército de técnicos revisa y sustituye constantemente las válvulas, que se funden a razón de treinta por hora.

Su tecnología electrónica le permite no tener más partes móviles que las de entrada y salida, puesto que todas las operaciones se realizan electrónicamente. Esto posibilita que sea 300 veces más rápido que el *Mark I*: Realiza sumas en 0.6 ms y divisiones en 15 ms. Almacena los números en decimal, y no en binario, a pesar de contar Eckert y Mauchly con la experiencia obtenida por Atanasoff con su *ABC*. Por último, las instrucciones no se almacenan en memoria, sino que se alimentan por medio de tableros de conexiones, que se modifican para

implementar el programa que se desee ejecutar.

El *ENIAC* será utilizado por los militares americanos hasta 1955, año en que será donado al Smithsonian Institute.

Bibliografía

-*Informática Básica*. Antonio Novel i Cabré. Manuales de Informática. Ed. Centre de Cálcul de la Universitat Politécnica de Catalunya.

-*Informática Básica. Ed. CEDED.*

-*Informática Básica. Serie Inf*ormática de Gestión. Ed. McGraw-Hill.

-*Enciclopedia Mi Computer.* Ed. Delta.

-*Informática*. Formación Profesional Administrativa. 2º Grado, 3er curso.

-*Introducción a la Informática*. Llorenç Guilera Agüera. Edunsa, Ediciones y Distribuciones Universitarias, S. A.

La precisión ante todo (III)

Continuando la saga, en este número listamos las primeras 1229 cifras exactas de la raíz cuadrada de dos. Por cierto, se muestran 1230 cifras pero la última está redondeada. En realidad no sé por qué lo digo, ya que nadie se iba a dar cuenta... Esta sección no sirve para nada, pero queda la mar de bien...

1.41421356237309504880168872420969807856967187537694807317667973799073247846210703885038753432764157273501384623091229702492483605585073721264412149709993583141322266592750559275579995050115278206057147010955997160597027453459686201472851741864088919860955232923048430871432145083976260362799525140798968725339654633180882964062061525835239505474575028775996172983557522033753185701135437460340849884716038689997069900481503054402779031645424782306849293691862158057846311159666871301301561856898723723528850926486124949771542183342042856860601468247207714358548741556570696776537202264854470158588016207584749226572260020855844665214583988939443709265918003113882464681570826301005948587040031864803421948972782906410450726368813137398552561173220402450912277002269411275736272804957381089675040183698683684507257993647290607629969413804756548237289971803268024744206292692485905218100445984215059112024944134172853147810580360337107730918286931471017111168391658172688941975871658215212822951848847208969463386289156288276595263514054226765323969461751291602408715510135150455381287560052631468017127402653969470240300517495318862925631385188163478001569369176881852378684052287837629389214300655869568685964595155501644724509 8