EJERCICIOS DE PROGRAMACIÓN EN FORTH

F. J. Gil Chica

diciembre, 2009

Capítulo 1

Enunciados

1.1. Consideraciones previas

Esta es una colección de ejercicios de programación en Forth. Abarcan desde manipulación básica de la pila hasta cálculo numérico y creo que son ilustrativos del lenguaje y de la forma en que se programa con él.

Respecto a lo primero, hemos omitido algunos temas más avanzados de los que podrían componerse ejercicios que involucrasen el uso de `immediate`, `postpone`, `catch` y `throw`, por ejemplo.

Respecto a lo segundo, observará el lector en primer lugar que hemos compuesto todos los ejercicios sin intervención de variables: todos los cálculos se basan en la pila. En general, esto es más elegante y ajustado a la filosofía del lenguaje, aunque reconocemos que algunos de los ejercicios hubiesen quedado más claros y limpios introduciendo variables locales. Pero no se olvide lo que son: ejercicios. También observará el lector que en general las palabras son muy pequeñas. Si reunimos el vocabulario contenido en este texto y lo compilamos (usamos *Gforth*) veremos que la media es inferior a 100 bytes por palabra. No obstante, hay algunas pocas de estructura muy lineal que son inusualmente largas para el lenguaje, aunque muy breves comparadas con el tamaño medio de las funciones en cualquier otro lenguaje. En estos pocos casos, es obvia la forma en que pueden factorizarse.

1.2. Manipulación de la Pila

Salvo indicación contraria, se suponen operaciones con números enteros.

1. Escribir una palabra que produzca el efecto en la pila `(a b -- a+1 b-1 a+b)`.

2. Escribir una palabra que produzca el efecto en la pila `(a b -- a*a a*b)`.

3. Dados tres números en la pila, (a, b, c), calcular $d = b^2 - 4ac$.

4. Escribir una palabra que tome dos números (a, b) de la pila y deje como resultado $(a - b)$ y $(a + b)$.

5. Escribir una palabra que tome tres números (a, b, c) de la pila y deje como resultado $a^2 + b^2 + c^2$.

6. Escribir una palabra que tome un número n de la pila y deje como resultado $n^2 - 5n + 6$.

7. Escribir una palabra que tome una dirección de la pila y deje la misma dirección incrementada en una celda y el valor contenido en la dirección original.

8. Escribir una palabra que tome dos direcciones de la pila y deje ambas direcciones incrementadas en una unidad y los contenidos de las direcciones originales `( d1 d2 -- d1+1 d2+1 v1 v2)`.

9. Escribir una palabra que tome dos direcciones de la pila e intercambie sus contenidos.

10. Escribir una palabra que produzca el efecto en la pila `(a b>0 -- a*b)` **sin** usar la multiplicación. Una multiplicación no es más que una suma repetida.

11. Escribir una palabra que produzca el efecto en la pila `(a b -- a/b a%b)` es decir, que tome dividendo y divisor y deje cociente y resto, **sin** usar la división. Una división no es más que una resta repetida.

12. Implementar las palabras `rot` y `-rot`.

13. Implementar la palabra `2swap` cuyo efecto sobre la pila es `(a b c d -- c d a b)`.

14. Implementar la palabra `tuck`, cuyo diagrama de pila es `(a b -- b a b)`.

15. Implementar `2drop`, que elimina los dos elementos superiores de la pila. Implementar la palabra `2dup`, cuyo diagrama de pila es `(a b -- a b a b)`. Escribir la palabra `2over` cuyo efecto sobre la pila es `( a b c d -- a b c d a b`.

16. Implementar la palabra `3dup`, cuyo efecto sobre la pila es `( a b c -- a b c a b c)`.

17. Escribir una palabra que elimine todo elemento de la pila *sólo en el caso de que la pila contenga algún elemento que eliminar.*

18. Escribir una palabra llamada `pick` (totalmente desaconsejable su uso) que trate a la pila como un vector, copiando el elemento n que se indique en la cima de la pila. n numera los elementos de la pila desde el más reciente, 0, al más profundo, y no se cuenta a sí mismo (ej. si la pila contiene 1, 2 y 3, `1 2 3 2 pick` tiene como resultado `1 2 3 1`)

1.3. Memoria

1. Escribir una palabra que recupera el valor contenido en una dirección de memoria, dejando en la pila la dirección original incrementada en una unidad (o en una celda), y el valor recuperado.

2. Escribir una palabra que intercambia los valores contenidos en dos direcciones de memoria.

3. Escribir una palabra que copia en la pila el último elemento apilado en la pila de retorno.

4. Escribir una palabra que toma de la pila una dirección origen y una dirección destino, copia el valor contenido en la dirección origen en la dirección destino y devuelve en la pila las direcciones originales incrementadas en una unidad.

5. Escribir una palabra que toma de la pila dos direcciones, una origen y otra destino, y un número n, copiando n bytes a partir de la dirección origen en las n posiciones correlativas a partir de la dirección de destino.

6. Escribir una palabra que tome de la pila dos direcciones y devuelva: ambas direcciones incrementadas en una unidad y los valores contenidos en las direcciones originales.

7. Escribir una palabra que desplaza una porción de memoria el número de posiciones que se indique hacia «arriba». Idem hacia «abajo».

8. Un bloque de memoria se especifica mediante un dirección base y un tamaño en bytes. Escribir dos palabras que esperen en la pila la especificación de un bloque y un entero que represente cuántos bytes ese

bloque ha de ser desplazado, y efectúe el desplazamiento. El desplazamiento puede hacerse «hacia arriba» o «hacia abajo», de ahí las dos palabras.

1.4. Números enteros

1. Escribir una palabra que tome dos números de la pila y deje el máximo común divisor de ambos. Si a y b son estos números y suponemos que $a > b$, úsese la propiedad de que $mod(a, b) = mod(b, r)$, donde r es el resto de la división entera a/b. Por aplicación repetida de esta propiedad, puede demostrarse que el valor buscado es el del primer argumento cuando el segundo se hace nulo.

2. Dos números se llaman primos si el máximo común divisor a ambos es 1. Escribir una palabra que tome dos números y deje verdadero o falso según sean primos o no.

3. Escribir una palabra que busque el mínimo divisor (distinto de 1) de un número dado en la pila.

4. Basándose en el ejercicio anterior, escribir una palabra que deje un *flag* en la pila indicando si el número depositado en ella es o no primo: `(n -- flag)`.

5. Imprimir todos los números primos entre dos dados.

6. Escribir una palabra que produzca la descomposición en factores primos de un número dado.

7. Dado un número n y una base b, encontrar la potencia p tal que $b^p < n <= b^{p+1}$.

8. Dada una función entera de argumento entero $f(n)$, encontrar la sumatoria

$$\sum_{i=0}^{N} f(i) \tag{1.1}$$

   Se esperan en la pila N y la dirección de f.

9. Dados dos enteros en la pila representando un racional (numerador, denominador), encontrar el máximo común divisor de ambos y dividirlos por él. El resultado es la fracción canónica del número original (ej. 9/27 se reduce a 1/3).

10. Escribir palabras `qrot`, `-qrot`, `qtuck` y `qnip` aptas para manipular números racionales (cada número racional ocupa dos celdas de la pila).

11. Un número racional se representa mediante dos enteros depositados en la pila: el numerador y el denominador. Escribir palabras para la suma, resta, multiplicación y división de racionales.

12. Dados dos números en la pila, n y k, calcular el coeficiente binomial

$$C_{n,k} = \frac{n!}{k!(n-k)!} \tag{1.2}$$

13. Un algoritmo sencillo para generar números aleatorios parte de una semilla n_0 y calcula según

$$n_{j+1} = (an_j + c)\,\%m \tag{1.3}$$

En concreto, cuando $a = 899$, $c = 0$ y $m = 32768$ se generan números en el intervalo $[0, 32767]$. Escribir un generador de números aleatorios.

14. Dado un número p encontrar el mayor $n <= \sqrt{p}$.

15. Dado un número p y una raíz aproximada $n <= \sqrt{p}$ encontrar la representación racional de una mejor aproximación de acuerdo con la fórmula:

$$n_{k+1} = \frac{1}{2}\left(n_k + \frac{p}{n_k}\right) \tag{1.4}$$

16. Dado un número entero, escribir una palabra que imprima su representación en binario.

17. Dado un entero, escribir una palabra que imprima su representación en hexadecimal.

18. Dado un entero n, escribir una palabra que consulte el valor del bit p, dejando un 0 o 1 en la pila según que dicho bit esté a 0 o 1.

19. Dado un entero n y un número p, escribir palabras para activar y desactivar el bit p de n.

20. Un número complejo, con partes reales e imaginaria enteras, se representa en la pila mediante dos enteros: parte real y parte imaginaria. Escribir palabras para sumar, restar, multiplicar y obtener una potencia de un complejo.

21. Dados dos complejos $a + bj$ y $c + dj$, se define su razón como

$$\frac{a + bj}{c + dj} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2} j \tag{1.5}$$

Implementar una palabra que efectúa la división. Puesto que las partes reales e imaginaria ya no son enteros, sino racionales, la palabra dejará en la pila primero los numeradores, real e imaginario, y luego el denominador, común.

1.5. Vectores

Consideramos por simplicidad vectores de números enteros. La versión para números reales es obvia. Nos apoyaremos en el tipo `vector`. Al escribir

```
10 vector X
```

Se espera que sea creado en el diccionario un vector de 10 elementos. Un vector de 10 (n) elementos ocupa 11 (n+1) celdas en el diccionario, pues la primera de ellas contiene el tamaño del vector. En tiempo de ejecución, la palabra `X` creada anteriormente se espera que deje en la pila la dirección del primer elemento y el número de elementos. Para los diagramas de pila, usaremos las letras `i` para un índice, `v` para un valor, `d` para una dirección y `N` para el número de elementos del vector.

1. Escribir una palabra, `vector` que cree en el diccionario un vector con el número de elementos depositado en la pila y la estructura explicada.

2. Escribir una palabra `v!` que espere en la pila un valor, la posición donde quiere guardarse, el tamaño del vector y su dirección. (ej. `4 10 X v!` almacena el valor 4 en la posición 10 del vector `X`; las posiciones se indexan desde la 0)

3. Escribir una palabra `v@` que espere en la pila el elemento del vector que quiere recuperarse, el tamaño de éste y su dirección, y deje en la pila el valor buscado.

4. Escribir una palabra que imprima en columna los elementos de un vector. Se esperan su tamaño y dirección. Escribir una palabra que imprima un rango de elementos de un vector.

5. Dada una pila que contiene una serie de argumentos $n_1...n_p$, el número m de ellos y el tamaño y dirección de un vector, escribir una palabra que escribe los valores n_i sucesivamente a partir de la dirección base. Obviamente, los elementos del vector quedarán ordenados inversamente a como fueron apilados.

6. Dado un vector, calcular la suma de todos sus elementos.

7. Dado un vector, escribir una palabra que averigüe su valor máximo. Idem para el valor mínimo.

8. Dado un valor, contenido en un vector, encontrar la posición en que se encuentra.

9. Dado un valor, averigüar si se encuentra en un vector dado.

10. Calcular la media de un vector.

11. Escribir una palabra que rellene con ceros todos los elementos de un vector. Generalizarlo para cualquier valor distinto de cero.

12. Escribir una palabra que aplique una función cualquiera a cada uno de los elementos de un vector.

13. Escribir una palabra que escriba en cada elemento de un vector el resultado de aplicar una función cualquiera a su índice (ej. cada elemento i se sustituye por $f(i)$).

14. Escribir una función que rellene un vector con valores aleatorios proporcionados por la función desarrollada en la sección anterior.

15. Escribir una función que devuelva verdadero o falso según que un valor se encuentre o no entre los elementos de un vector.

16. Dado un vector y dos índices, intercambiar los valores correspondientes.

17. Escribir el algoritmo de la burbuja para ordenar los elementos de un vector de menor a mayor.

18. Escribir una palabra que escriba en orden inverso los elementos de un vector.

19. Dado un vector, sustituir cada elemento en la posición i por la suma de todos los elementos anteriores hasta el i incluido.

1.6. Cadenas

Una cadena de caracteres no es más que un vector de caracteres. Cuando se reserva espacio para una cadena de caracteres, la primera celda se destina a guardar el tamaño reservado. A diferencia de lo que ocurre con los vectores, es interesante también saber cual es el tamaño efectivo de las cadenas que se almacenen. Por tanto, definiremos una cadena como una estructura que consta de tres campos: una celda con el tamaño máximo; una celda con el tamaño ocupado y una secuencia de caracteres. En tiempo de ejecución, la llamada a una cadena ha de dejar en la pila el tamaño máximo, el efectivo y la dirección del primer carácter.

1. Escribir una palabra para crear una cadena de caracteres.

2. Escribir una palabra para imprimir en pantalla una cadena de caracteres.

3. Escribir una palabra que lee caracteres de teclado y forma con ellos una cadena, almacenada en la variable de tipo cadena que se indique.

4. Dadas dos cadenas, copiar la primera en la segunda.

5. Dadas dos cadenas, copiar la segunda a continuación de la primera, si eso es posible.

6. Escribir una palabra que compara dos cadenas.

7. Escribir una palabra que compara el mismo intervalo de dos cadenas.

8. Escribir una palabra que comprueba si una cadena (a la que nos referimos como la subcadena) dada se encuentra contenida en otra. Escribir una segunda versión de la palabra anterior que en lugar de devolver un *flag* devuelve -1 si la subcadena no está contenida en la cadena o la posición del primer carácter de la subcadena dentro de la cadena si la primera está contenida en la segunda. Se da por supuesto que la subcadena es de longitud inferior a la cadena.

9. Escribir palabras para, dada una cadena, transformar su texto a mayúsculas o a minúsculas.

10. Escribir una palabra que devuelva la posición en que aparece por primera vez un carácter dado, o -1 si ese carácter no aparece en la cadena

11. Escribir una palabra para eliminar de una cadena el carácter que se encuentra en una posición dada.

12. Escribir una palabra para insertar en una cadena un carácter dado en la posición que se indique.

13. Escribir una palabra que elimine los espacios en blanco que se encuentren al principio de la cadena.

14. Escribir una palabra que elimine los espacios en blanco que se encuentren al final de la cadena.

15. Una cadena está formada por palabras separadas por espacios en blanco. Identificar las direcciones de comienzo de cada palabra.

16. Escribir una palabra que deje una bandera en la pila indicando si una cadena determinada es vacía (contiene sólo espacios en blanco) o no.

17. Escribir una palabra que elimina espacios en blanco redundantes entre palabras. Es decir, que deja uno y sólo un espacio en blanco entre una palabra y la siguiente.

18. Escribir una palabra que espere en la pila un entero que representa una posición dentro de una cadena y la dirección de comienzo de esa cadena, y que deje en la pila el número de caracteres distintos de espacio que hay entre la posición dada y el siguiente espacio en blanco. Apta para encontrar la longitud de cada palabra dentro de una cadena.

19. Dada una cadena, escribir palabras para ajustarla a izquierda, derecha y centro. Se entiende que el ajuste se produce sin que cambie la longitud efectiva de la cadena.

1.7. Matrices

Un vector es suficiente para representar una matriz, de dos o más dimensiones. Sin embargo, conviene ocultar la conversión entre coordenadas de (fila,columna) y dirección lineal dentro del vector, y conviene así mismo que el tipo de dato «matriz» lleve asociada información sobre el número de filas y columnas, ya que un mismo vector de 18 elementos puede igualmente representar una matriz de 3x6 y una de 2x9 elementos. Por simplicidad, consideraremos matrices de enteros. Sin dificultad pueden escribirse versiones para matrices de reales o racionales.

1. Escribir una palabra que permita crear una matriz. Las dimensiones estarán previamente depositadas en la pila. En el diccionario se reservará espacio para estos dos números y a continuación para el total de

elementos. En tiempo de ejecución se depositará en la pila la dirección del primer elemento. Se escribirán palabras específicas para obtener el número de filas y el número de columnas de una matriz.

2. Escribir una palabra `m!` que espere en la pila un valor a almacenar, una pareja (i, j) y una matriz y almacene dicho valor en la posición correspondiente de la matriz.

3. Escribir una palabra `m@` que tome de la pila un valor de fila y columna, (i, j), y una matriz y recupere el elemento correspondiente.

4. Escribir una palabra `m.` que imprima en pantalla los elementos de la matriz, separando las filas en líneas distintas.

5. Escribir una palabra que rellene una matriz con el valor 0.

6. Escribir una palabra que aplique a cada elemento de una matriz x la función $f(x)$

7. Escribir una palabra que espere en la pila una matriz y la dirección de una función $f(i, j)$ y escriba en cada elemento (i, j) de la matriz el valor $f(i, j)$.

8. Escribir una palabra que espere en la pila una matriz y devuelva el valor máximo y la fila y columna en que se encuentra. Igual para el valor mínimo.

9. Escribir una palabra que rellene una matriz con valores aleatorios.

10. Escribir palabras para intercambiar dos filas y dos columnas de una matriz.

11. Escribir una palabra que trasponga una matriz (cuadrada), intercambiando cada elemento (i, j) por el elemento (j, i).

12. Escribir una palabra para multiplicar una fila de una matriz por un valor determinado. Se esperan en la pila el valor que multiplicará, la fila y la matriz. Igual para una columna.

13. Escribir una palabra que sustituya una fila por el resultado de sumarle otra. Es decir, si las filas a sumar son la i y la j, para toda columna k, $a_{ik} = a_{ik} + a_{jk}$.

1.8. Cálculo numérico

En esta sección, se suponen números reales.

1. Dada una función real de variable real, escribir una palabra que imprima en pantalla una tabla de valores de esa función. Se esperan en la pila el incremento en el argumento, los extremos inferior y superior del intervalo y la dirección de la función.

2. Escribir palabras para crear vectores de números reales, recuperar elementos y almacenar valores en posiciones dadas.

3. Escribir una palabra que espere en la pila un valor inferior de la variable independiente, un incremento para esa variable, la dirección de una función y un vector de reales. Almacenar en ese vector el valor de la función para sucesivos valores de la variable independiente, separados por el incremento establecido, es decir, si a es el valor inicial para la variable independiente y h el incremento, el vector almacenará $f(a)$, $f(a+h)$, $f(a+2h)$, etc.

4. Dado el conjunto de valores reales $x_0, ..., x_n$ el entero $n{+}1$ y un vector de reales, escribir una palabra que escriba los elementos x_i en las posiciones sucesivas del vector.

5. Calcular la función exponencial para un argumento x mediante la serie

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + ... \tag{1.6}$$

6. Dada una función $f(x)$ y dos valores $x = a$ y $x = b$ de forma que existe una raíz de $f(x) = 0$ en el intervalo $[a, b]$, encontrar dicha raíz por el método de la bisección.

7. Usar el ejercicio anterior para escribir una palabra que tome un intervalo de la recta real y la dirección de una función y devuelva todas las raíces reales de la función en el intervalo dado.

8. Escribir una versión que use números reales del algoritmo explicado en la sección sobre números enteros para calcular la raíz cuadrada de un $x >= 1$.

9. Dado un vector, dos argumentos $a < b$ y la dirección de una función, calcular la integral

$$\int_a^b f(x)dx \tag{1.7}$$

por el método de los trapecios con la resolución que indique la dimensión del vector.

10. Dados dos vectores de reales que almacenan series de valores x_i y y_i, calcular los coeficientes (a, b) de la recta $y(x) = ax+b$ que ajusta mejor los datos (x_i, y_i) (ajuste de mínimos cuadrados).

11. Escribir una palabra que, dados en la pila un valor perteneciente al dominio de una función y la propia función, calcule numéricamente el valor de la derivada de la función en el punto dado.

12. Podemos representar un polinomio mediante un vector que almacene sus coeficientes para las potencias crecientes de la variable independiente. Por ejemplo, el vector `[ 0.2e 0.8e -1.3e ]` representa al polinomio $p(x) = 0{,}2 + 0{,}8x - 1{,}3x^2$. Escribir una palabra que espere un valor de x y un vector que representa un polinomio y calcule el valor del polinomio en el punto.

13. Considerando los elementos de un vector de reales como los coeficientes de un polinomio, sobreescribirlo con su derivada.

14. Igual, pero con la integral.

15. Dado un vector que contiene los coeficientes reales de un polinomio $p(x)$ y dos números (a, b), obtener

$$\int_a^b p(x)dx \tag{1.8}$$

16. Integrar por el método de Runge-Kutta de cuarto orden la ecuación diferencial sencilla:

$$\frac{dx}{dt} = f(t) \tag{1.9}$$

dados unos valores iniciales (t_0, x_0).

17. Integrar por el método de Runge-Kutta de cuarto orden la ecuación diferencial sencilla:

$$\frac{dx}{dt} = f(x) \tag{1.10}$$

18. La integral doble

$$\int_0^1 \int_0^1 f(x,y)dxdy \tag{1.11}$$

puede resolverse aproximadamente mediante el método de Montecarlo. Si se genera una gran cantidad N de parejas de números aleatorios (x_i, y_i) en el intervalo $[0,1]$, la integral es aproximadamente

$$\frac{1}{N}\sum_i f(x_i, y_i) \tag{1.12}$$

Escribir una palabra que tome la dirección de $f(x,y)$ y devuelva el valor de la integral.

1.9. Archivos

Cuando tratemos con la escritura y lecturas de valores numéricos en archivos, supondremos que estos números son enteros. Supondremos asímismo también que los archivos son archivos textuales, no binarios, de forma que los valores numéricos se almacenan como las cadenas que representan a los números. Serán útiles las palabras para cadenas desarrolladas anteriormente.

1. Escribir una palabra que abra un archivo de texto, lo lea línea a línea e imprima en pantalla cada línea.

2. Escribir una versión de la palabra anterior que numera cada línea antes de imprimirla.

3. Escribir una palabra que acepta cadenas desde teclado, terminadas con la tecla `Intro`, y las escribe en un archivo.

4. Un archivo consta de una sola columna de números enteros. Escribir una palabra que los lee uno a uno y rellena con ellos un vector.

5. Igual, pero con dos columnas, de tal forma que la primera columna rellena un vector y la segunda otro.

6. Escribir una palabra que lee todas las líneas de un archivo e imprime aquellas que contienen una secuencia dada de caracteres.

7. Dado un archivo de texto que consta de una serie de columnas, escribir una palabra que toma de la pila los números de dos de esas columnas y las imprime en pantalla.

8. Supóngase un archivo de texto que contiene columnas. Guardar en un vector la anchura máxima de cada columna.

9. Usar el resultado anterior para, dado un archivo que contiene columnas de forma que la anchura de cada columna es variable de una línea a otra, tomar el archivo y construir otro donde la anchura de cada columna sea uniforme.

1.10. Teclado y pantalla

Obviamente, esta sección es dependiente del sistema sobre el que se trabaje. Nosotros la hemos desarrollado sobre un terminal Linux.

1. Escribir una palabra que, dado un carácter y una coordenadas de fila y columna en la pila, imprima en pantalla ese carácter en la posición especificada.

2. Escribir una palabra que imprima en pantalla una línea con un número especificado de copias de un carácter especificado.

3. Escribir una palabra que imprima una tabla con los caracteres ASCII.

4. Escribir una palabra que dibuje en pantalla un recuadro de anchura y altura especificadas. Se usarán los caracteres '-' para los trazos horizontales, '—' para los verticales y '+' para las esquinas. Se esperan en la pila las coordenadas de la esquina superior izquierda, anchura y altura.

5. Escribir una rutina que dado un carácter '*' en el centro de un recuadro, dibujado con la palabra anterior, use las teclas 'A','S','W','Z' para moverlo a izquierda, derecha, arriba y abajo, sin que abandone los límites del recuadro. Usar la tecla 'Q' para abandonar el programa.

6. Escribir una palabra que envíe al terminal la secuencia ANSI que activa el video inverso.

7. Escribir una palabra que rellene una cadena con los 16 caracteres [A,...,O,-]. Los 15 primeros se tomarán en orden aleatorio. El objeto es dibujar en pantalla una matriz como la de la figura:

```
+---+---+---+---+
| G | B | I | J |
+---+---+---+---+
| M | H | L | C |
+---+---+---+---+
| N | A | E | D |
+---+---+---+---+
| F | O | K | - |
+---+---+---+---+
```

y jugar con el teclado a desplazar las letras, aprovechando el «hueco» indicado por '-', hasta re-ordenadarlas en orden alfabético.

8. Dado un carácter en la pila, escribir un programa que produzca en pantalla una versión «macro» de ese carácter, de tamaño 10 filas por 8 columnas. Por ejemplo, para el carácter 'P'

```
##########
##      ##
##      ##
##      ##
##      ##
##########
##
##
##
##
```

9. Escribir una rutina que imprima una cadena en la parte superior de la pantalla. A continuación, se señalará con el carácter '*' el primero de la cadena, imprimiéndolo en la línea inferior. Se usarán las teclas 'A' y 'S' para avanzar y retroceder a lo largo de la cadena, actualizando la señal '*' e imprimiendo en formato «macro» el carácter señalado, al modo que muestra la figura.

```
Muchos programadores, ignoran la existencia de Forth
       *

##########
##      ##
##      ##
##      ##
##      ##
##########
##
##
##
##
```

1.11. Estructuras de datos

1.11.1. Pilas y colas

1. Un usuario puede construir una pila para su propio uso reservando mediante `allot` un conjunto de celdas, de las cuales la primera es un índice que indica la siguiente posición libre en la pila y la segunda el número máximo de elementos que puede contener. Escribir una palabra para crear pilas de usuario en la forma

    ```
    20 pila X  \ crea pila de usuario X de 2+18 celdas
    ```

    En tiempo de creación, se reserva el espacio y se establece a 1 el valor de la primera celda (índice 0). En tiempo de ejecución, se deposita en la pila la dirección de la pila de usuario. Una pila no es más que un vector usado de una forma particular.

2. Escribir palabras `p+` y `p-` para apilar y desapilar valores en la pila de usuario.

3. Escribir palabras `p-drop` y `p-swap` y `p-dup` que actúen sobre la pila de usuario.

4. Escribir palabras `p-r>` y `p-r<` que actúen sobre la pila de usuario. Escribir palabras que traspasen elementos entre dos pilas de usuario, desapilando un elemento de una pila y apilándolo en la otra.

5. La primera celda de una pila se usa como índice que va creciendo y decreciendo según se apilan o desapilan elementos. En una cola FIFO no se extrae en primer lugar el último elemento apilado, sino el primero. Los elementos se apilan «por arriba» pero se extraen «por debajo». Ahora bien, para no agotar rápidamente el espacio en la cola, cuando se extrae un elemento sería preciso desplazar «hacia abajo» todos los demás. Como esto es costoso, se prefiere implementar como una cola circular. Una cola circular es un vector cuyas dos primeras celdas se reservan. La primera guarda el índice de la siguiente posición libre para encolar. La segunda, la posición del siguiente elemento a desencolar. Tras cada operación de encolar o desencolar, el índice correspondiente se incrementa. Si la cola tiene N elementos, de 0 a $N-1$, cuando un índice vale ya $N-1$ se establece a 2. La cola se encuentra llena cuando el índice de escritura está justo detrás del de lectura, y vacía cuando el índice de lectura se encuentra justo detrás del de escritura. En nuestra implementación, reservaremos no dos sino cuatro celdas: punteros de lectura y escritura, número de elementos que contiene la cola en cada momento y número máximo de elementos.

   Escribir una palabra para crear una cola FIFO de usuario, reservando un número dado de celdas,

   ```
   20 fifo Z \ crea una cola FIFO de 4+16 elementos
   ```

6. Escribir el conjunto de palabras que permita gestionar la cola FIFO.

1.11.2. Memoria dinámica

Muchas estructuras de datos como las listas y los árboles se basan en punteros, que a su vez presuponen memoria dinámica. Lenguajes como C y Pascal permiten declarar punteros, asignarles memoria y liberar esa memoria cuando ya no es preciso. Otros lenguajes como Java o Lisp hacen uso intensivo de punteros, pero no son una característica del lenguaje sino uno de los elementos mediante los que el lenguaje se implementa. Toda la complejidad queda oculta para el programador. De la misma forma, toda la flexibilidad y potencia también quedan inaccesibles.

Todos los sistemas ofrecen memoria dinámica, pero nosotros usaremos aquí una aproximación diferente: un programa puede reservar un trozo del diccionario y gestionarlo dinámicamente, tomando y liberando memoria cuando sea preciso. Pero no directamente, sino a través de un *gestor de memoria* que ofrece una interfaz y consta de las palabras necesarias para gestionar internamente la asignación y liberación de memoria, desfragmentación de la misma, gestión de errores, etc.

En la literatura anglosajona, a un trozo de memoria que se gestionará de forma dinámica, tomando o liberando porciones de él, se le llama *heap*, que a veces se traduce como *montón*. Nos parecen más adecuados nombres como *depósito* o *almacén*. Le llamaremos de esta segunda forma.

Hay muchas formas de gestionar un almacén, y puede consultarse al respecto cualquier libro de sistemas operativos. Nuestra aproximación será muy, muy simple: tanto de entender como de implementar. Sin embargo, lo que obtendremos al final será un auténtico gestor de memoria dinámica, y no otra cosa. En resumen, esta sección contiene un único ejercicio que se resolverá en un conjunto de palabras que explicaremos a medida que las vayamos escribiendo. Así, más que un ejercicio éste es un pequeño proyecto donde se hará evidente la metodología *de abajo arriba* propia de Forth.

1. Escribir un gestor de memoria dinámica que reserve y administre un almacén. Un programa deberá poder declarar el almacén especificando su nombre y el tamaño que será administrado: `10000 almacen A`.

Capítulo 2

Soluciones

Respecto a las soluciones que se proponen, se tendrán en cuenta algunas cosas: a) que las soluciones no son únicas; b) que de las posibles soluciones a un ejercicio, la nuestra será razonable, pero dudamos de que sea óptima; c) algunas palabras podrían sin duda factorizarse.

Respecto a la notación, los comentarios de pila usan las letras para distintos tipos de datos: `a,b,c,m,n,k` para números enteros; `d,e` para direcciones de memoria; `u,v` para valores recuperados de las direcciones de memoria; `x,y,z` para números reales; `xt` para el *execution token* de una palabra; `f` es un *flag*, o resultado de una prueba (verdadero o falso); `p,q,r` son números racionales o complejos, de tal forma que representan, cada uno, dos enteros en la pila: (numerador,denominador) o (parte real, parte imaginaria)

2.1. Manipulación de la Pila

1. `: p1 1- swap 1+ swap 2dup + ;`
2. `: p2 over * swap dup * swap ;`
3. `: p3 swap dup * -rot 4 * * - ;`
4. `: p4 2dup - -rot + ;`
5. `: p5 dup * swap dup * + swap dup * + ;`
6. `: p6 dup dup * swap 5 * - 6 + ;`
7. `: p7 dup @ swap cell+ swap ;`
8. `: p8 2dup @ swap @ swap >r >r 1+ swap 1+ swap r>  r> ;`

9. ```
   : p9 2dup @ swap @ rot ! swap ! ;
   ```

10. ```
    : p10 0 swap 0 do over + loop nip ;
    ```

11. ```
    : p11 0 -rot
         begin
             2dup >=
         while
             tuck - swap rot 1+ -rot
         repeat
         drop ;
    ```

12. ```
    : rot ( a b c -- b c a) >r swap r> swap ;
    : -rot ( a b c -- c a b) swap >r swap r> ;
    ```

13. ```
    : 2swap ( a b c m -- c m a b) >r -rot r>  -rot ;
    ```

14. ```
    : tuck ( a b -- b a b) dup -rot ;
    ```

15. ```
    : 2drop ( a b -- ) drop drop ;
    : 2dup ( a b -- a b a b) over over ;
    : 2over ( a b m n -- a b m n a b )
        2swap 2dup >r >r 2swap r> r> ;
    ```

16. ```
    : 3dup ( a b c -- a b c a b c)
        >r 2dup r> dup >r -rot r> ;
    ```

17. ```
    : cls depth dup 0= if drop else 0 do drop loop then ;
    ```

18. Es obvio que los elementos que se encuentran por encima en la pila de aquel del que se quiere extraer una copia han de ser movidos a otros sitio, y luego devueltos. ¿Dónde podrían moverse? Un lugar puede ser la pila de retorno. Pero como la operación de retirar n elementos ha de consistir en un bucle que retire un elemento cada vez y este bucle ha de contar con la pila de retorno para ejecutarse, ésta pila no parece el mejor lugar, aunque es posible usarla. Más natural es usar el diccionario, a partir de la primera posición libre, cuya dirección es devuelta por la palabra `here`. Así pues, `pick` consta esencialmente de dos pasos: a) retirar n elementos de la pila y colocarlos sucesivamente a partir de la primera posición libre en el diccionario y b) recuperar estos elementos uno o uno llevando en cada paso una copia del elemento que se quiere replicar a la parte de arriba de la pila, de tal forma que al terminar este segundo bucle se encuentre sobre todos los demás. `quitar-n` espera en la pila una dirección a partir de la cual colocar elementos y el número
    ```

de elementos que se quieren mover. `devolver-n` una dirección desde la que empezar y el número de elementos que se quieren devolver a la pila (en orden descendente desde la dirección inicial).

```
: cell+ 4 + ;  \ una celda tiene 4 bytes
: cell- 4 - ;
: quitar-n ( d n -- d') 0 do 2dup ! nip cell+ loop ;
: devolver-n ( d n -- d') 0 do dup @ -rot cell- loop ;
: pick dup >r here swap quitar-n
    cell- over swap r> devolver-n drop ;
```

2.2. Memoria

1.
```
: @+c ( d -- v d') dup @ swap cell+ swap ;
: @+1 ( d -- v d') dup c@ swap 1+ swap ;
```

2.
```
: exch ( d d' --)
    2dup @ swap @ rot ! swap ! ;
```

3.
```
: r@ r> dup >r ;
```

4.
```
: inc2 ( a b -- a+1 b+1)
    1+ swap 1+ swap ;
: dec2 ( a b -- a-1 b-1) \ para cumplir con la simetr\'{\i}a
    1- swap 1- swap ;
: cp1 ( o d -- o+1 d+1)
    2dup swap c@ swap c! inc2 ;
```

5.
```
: cp* ( o d n -- )
    0 do cp1 loop drop drop ;
```

6. Una versión para caracteres y otra para celdas.

```
: 2@+1 ( d d' -- d+1 d'+1 v v')
    2dup inc2 2swap c@ swap c@ swap ;
: 2@+c  ( d d' -- d+c d'+c v v')
    2dup 1 cells + swap 1 cells + swap
    2swap @ swap @ swap ;
```

7. Escribiremos las palabras `mov+` y `mov-` para efectuar desplazamientos de bloques de memoria, «hacia arriba» y «hacia abajo». Puesto que la

posición final puede solaparse con la posición inicial, en los desplazamientos «hacia arriba» se desplazarán bytes desde el último del bloque hacia atrás, hasta la dirección base, que será el último byte en desplazarse. Cuando el desplazamiento sea «hacia abajo» se procederá desplazando desde el primer byte del bloque hasta el último. Obsérvese que ya disponemos de la palabra `cp*`, pero que ésta no considera los posibles solapes.

```
: mov+ ( d n p --)
    >r tuck 1- + r> rot
    0 do
        2dup over +
        swap c@ swap c!
        swap 1- swap
    loop drop drop ;

: mov- ( d n p --)
    swap >r
    over swap -
    r> 0 do
        cp1
    loop drop drop ;
```

Obsérvese cómo en `mov-` las dos primeras líneas se ocupan sólo de preparar una dirección origen y una dirección destino. Esas direcciones son usadas por una palabra que ya teníamos: `cp1`, que efectúa la copia de un byte y actualiza, incrementándolas, las dos direcciones. Se ve que, en comparación `mov+` es notablemente más complicada, y que podría reescribirse si dispusiésemos de una palabra análoga a `cp1` que simplemente decrementase las direcciones en lugar de incrementarlas. Pero es que además, si así lo hiciésemos, podríamos implementar dos versiones de `cp*` y basar en ellas las dos versiones de `mov`.

Este es un buen momento para poner de manifiesto una de las características de la programación en Forth, queremos decir, de la *buena* programación en Forth. Y es: factorizar siempre que sea posible. No nos ha sido difícil escribir `mov+`, pero hemos advertido que puede hacerse mejor y que para ello precisamos otra palabra que, no siendo específica al problema que tenemos, podrá usarse en el futuro para factorizar otras palabras.

Y es que en Forth una palabra compleja puede descomponerse, factorizarse, en palabras más sencillas, y esas palabras, si son lo bastante

sencillas, pueden quedar desligadas del problema original y convertidas en piezas aptas para ser usadas en problemas distintos.

Por consiguiente: nunca debemos desaprovechar la oportunidad de hacer mejor las cosas, ya que *no es suficiente con que funcionen*, sino que han de estar escritas correctamente, es decir, todo lo cerca del ideal como sea posible. Nuestra impericia o pereza no alteran esta regla [1] .

He aquí el resultado de reconsiderar algunas de las palabras ya escritas a la luz del problema planteado en este ejercicio:

```
: cp1+ ( o d -- o+1 d+1)
    2dup swap c@ swap c! inc2 ;
: cp1- ( o d -- o+1 d+1)
    2dup swap c@ swap c! dec2 ;
: cp*+ ( o d n -- )
    0 do cp1+ loop drop drop ;
: cp*- ( o d n -- )
    0 do cp1- loop drop drop ;
: mov+ ( d n p --)
    >r tuck + 1- dup r> + rot cp*- ;
: mov- ( d n p --)
    swap >r over swap - r> cp*+ ;
```

2.3. Números enteros

1. ```
   : max-cd ( a b -- mcd(a,b))
       ?dup if
           tuck mod recurse
       then ;
   ```

2. Este es un excelente ejemplo que demuestra la concisión de Forth. Un programador que piense en C, o que traslade código C a código Forth, puede escribir descuidadamente:

   ```
   : primos? ( a b -- f)
       max-cd dup 1 = if drop -1 else drop 0 then ;
   ```

   Pero, si lo piensa mejor, puede notar que `max-cd` ya deja un resultado en la pila, y aprovechar ese resultado:

[1] Por supuesto, la regla tampoco se altera por los errores o deficiencias en el código en que *yo* haya incurrido en *este* libro
   ```

```
: primos? ( a b -- f)
    max-cd 1 = if -1 else 0 then ;
```

En un tercer momento, por fin, puede caer en la cuenta de que el *flag* que necesita se obtiene ya de la compación anterior al condicional. Ahora ya es Forth. Antes era C:

```
: primos? ( a b -- f)
    max-cd 1 = ;
```

3.
```
: min-div ( a  -- b)
    2
    begin
        2dup mod 0<>
    while
        1 +
    repeat
    nip ;
```

4. Un número es primo si su mínimo divisor es él mismo:

```
: primo? ( n -- f) dup min-div = ;
```

5. Puesto que los primos son impares, dejemos en la pila como límite inferior un número impar y eso permitirá incrementar de dos en dos, acelerando la búsqueda. Por tanto, la palabra espera en la pila el límite superior y un límite inferior impar. Una versión:

```
: listar-primos ( a b --)
    cr
    begin
        2dup >
    while
        dup primo? if dup . cr then
        2 +
    repeat drop drop ;
```

Y otra

```
: listar-primos ( a b --)
    do
        I dup primo? if . cr else drop then
        2
   +loop ;
```

6. Una versión iterativa:

```
: factor ( n -- )
    cr
    begin
        dup min-div dup . cr 2dup <>
    while
        /
    repeat ;
```

Si x es el número que deseamos factorizar e y su mínimo divisor, entonces imprimimos y y aplicamos la factorización a x/y. Es decir, podemos escribir una versión recursiva:

```
: f0 dup primo?
    if
        . cr
    else
        dup min-div dup . cr / recurse
    then ;
: factor cr f0 ;
```

7. Suponemos que $n > 1$ y $b > 1$. Nos servimos de la función «elevado a», **:

```
: ** ( a b -- a^b ) dup 0=
    if
        drop drop 1
    else
        over swap 1- recurse *
    then ;
: n7 ( n b -- p )
    0
    begin
```

```
        1+
        3dup ** <
    until
    nip nip 1- ;
```

8. Suponemos que se encuentran en la pila primero el índice del último término de la suma y después la dirección de la función. Se introducen mediante `-rot` la suma parcial y el contador del bucle.

```
: n8 0 -rot 0 -rot
    swap 1+ 0 do
        2dup execute
        >r rot r> +
        -rot swap 1+ swap
    loop drop drop ;
```

9. `: reduce ( a b -- a' b') 2dup max-cd tuck / >r / r>;`

10.
```
: qrot ( p q r -- q r p) >r >r 2swap r> r> 2swap ;
: -qrot ( p q r -- r p q) 2swap >r >r 2swap r> r> ;
: qnip ( p q -- q) rot drop rot drop ;
: qtuck ( p q -- q p q) 2swap 2over ;
```

11.
```
: q+ ( p q -- p+q) rot 2dup * >r rot * -rot * + r> reduce ;
: q- ( p q -- p-q) swap negate swap q+ ;
: q* ( p q -- p*q) rot * >r * r> reduce ;
: q/ ( p q -- p/q) >r * swap r> * swap reduce ;
```

12. El problema aquí es que por el rápido crecimiento de la función factorial, el numerador puede rápidamente exceder el rango de los enteros. No tiene sentido calcular primero $n!$ para después dividir por $(n-k)!$. Así pues, calculamos la expresión como

$$C_{n,k} = \frac{n(n-1)...(n-k+1)}{k!} \tag{2.1}$$

El problema entonces se reduce a calcular el productorio de la forma

$$\prod_{j=0}^{p}(n-j) \tag{2.2}$$

```
: factorial ( n -- n!) dup 0=
    if
        drop 1
    else
        dup 1- recurse *
    then ;
: prod ( n k -- n(n-1)...(n-k))
    1 -rot 0 do tuck * swap 1- loop drop ;
: C(n,k) ( n k -- C(n,k))
    dup factorial -rot prod swap / ;
```

Esta implementación no considera el caso trivial $C_{n,0} = 1$.

13. La única duda aquí puede ser de dónde obtener una semilla. Nosotros usaremos los dos últimos dígitos del puntero `here` (más 1 para prevenir el caso en que obtengamos 0) Dada la semilla, en cada llamada a `rnd` queda el nuevo valor y una copia que puede usarse.

```
: rnd-seed ( -- a) here 100 mod 1+ ;
: rnd ( a -- b b) 899 * 32768 mod dup ;
```

14.
```
: nsqrt ( a -- a nsqrt(a))
    0
    begin
        1+ 2dup dup * <=
    until 1- ;
```

15. La palabra `nsqrt` deja p y n en la pila, por este orden. Dividiendo cada uno por 1 se obtiene la representación racional. Únicamente hay entonces que efectuar las operaciones racionales indicadas:

```
: qsqrt nsqrt 1 tuck qtuck q/ q+ 1 2 q* ;
```

16. La primera versión es sencilla e imprime la representación binaria comenzando por el bit de menor peso. La segunda versión calcula en primer lugar los bits, los guarda en la pila y a continuación los imprime en orden inverso, el bit más significativo primero.

```
: .b-> ( n --)
    begin
```

```
        dup 2 mod 48 + emit 2 / dup 0=
    until drop ;
: .b ( n -- ) 0 swap
    begin
        dup 2 mod -rot swap 1+ swap 2 / dup 0=
    until drop
    0 do
        48 + emit
    loop ;
```

17.
```
: .h ( n -- ) 0 swap
    begin
        dup 16 mod -rot swap 1+ swap 16 / dup 0=
    until drop
    0 do
        dup 9 > if 55 + emit else 48 + emit then
    loop ;
```

18.
```
: bit? ( n b -- f) 0 ?do 2 / loop 2 mod ;
```

19.
```
: bit-a-1 ( n b -- n') 1 swap lshift or ;
: bit-a-0 ( n b -- n')
    2dup bit? 1= if
        1 swap lshift xor
    then ;
```

20.
```
: c+ ( p q -- p+q) rot + -rot + swap ;
: c- ( p q -- p-q) rot swap - -rot - swap ;
: c*imag ( p q -- a) >r * swap r> * + ;
: c*real ( p q -- a) rot * -rot * swap - ;
: c* ( p q -- p*q) 2over 2over c*imag >r c*real r> ;
: c**( p n -- p^n) 1- 0 do 2dup c* loop ;
```

21.
```
: c/real ( p q -- a) rot * -rot * + ;
: c/imag ( p q -- a) >r * swap r> * - ;
: c/ ( p q -- p/q)
    2dup
    dup * swap dup * + >r \ denominador
    2over 2over
    c/imag >r c/real r> r> ;
```

2.4. Vectores

Representaremos con letras mayúsculas X, Y... a la pareja de valores tamaño y dirección del vector que se depositan en la pila en tiempo de ejecución al llamar a un vector por su nombre (sección `does>` de la palabra `vector`).

1.
```
: vector create dup , cells allot
         does> dup @ swap cell+ swap ;
```

2. Una primera versión elimina de la pila las dimensiones del vector, suma a la dirección donde comienza el desplazamiento y hace la asignación:

```
: v!a ( v i d N -- ) drop swap cells + ! ;
```

   Una segunda versión puede usar el tamaño del vector para asegurarse de que no se escribe fuera de límites:

```
: v! ( v i d N -- ) rot min cells + ! ;
```

3. Como en el ejercicio anterior, nos aseguramos de no querer leer posiciones más allá del límite del vector:

```
: v@ ( i d N -- v) rot min cells + @ ;
```

4.
```
: v. ( d N -- )
    cr 0 do
        dup @ . cr cell+
    loop drop ;
: v(.) ( a b d N -- )
    drop rot tuck cells +
    -rot swap cr do
        dup @ . cr cell+
    loop drop ;
```

5. Esta versión también es segura: almacena como máximo el número de elementos que puede contener el vector.

```
: v!! ( a1...aj m d N --)
    rot min 0 do
        tuck ! cell+
    loop drop ;
```

6. ```
   : v-sum ( d N -- suma)
       0 -rot 0 do
           tuck @ +  swap cell+
       loop drop ;
   ```

7. ```
   : v-max ( d N -- v )
       over @ -rot 1 do
           cell+ tuck @ max swap
       loop drop ;
   : v-min ( d N -- v )
       over @ -rot 1 do
           cell+ tuck @ min swap
       loop drop ;
   ```

8. ```
   : v-pos ( v d N -- i)
       drop 0 begin
           2dup cells + @
           swap 1+ swap
           >r rot dup >r -rot
           r> r> =
       until nip nip 1- ;
   ```

9. `: v-med ( d N -- a) 2dup v-sum swap / nip ;`

10. `: 0v!! ( d N -- ) 0 do dup 0 swap ! cell+ loop drop ;`

11. `: nv!! ( a d N -- ) 0 do 2dup ! cell+ loop drop drop ;`

12. ```
    : v-map ( xt d N -- )
        0 do
            2dup @ swap execute
            over ! cell+
        loop drop drop ;
    ```

13. ```
    : v-map-i ( xt d N -- )
        0 swap 0 do
            3dup tuck cells +
            -rot swap execute
            swap ! 1+
        loop drop drop drop ;
    ```

14. El problema de esta función es: que si se basa en las palabras `rnd-seed` y `rnd`, puesto que la segunda usa de la primera, si se desea rellenar varios vectores con valores aleatorios, la secuencia de números que se
    ```

obtengan será la misma para todos, en tanto que el diccionario pertenezca inalterado, pues como dijimos, `rnd-seed` toma los dos dígitos últimos de `here`. Por eso preferimos aquí suministrar manualmente una semilla cada vez.

```
: v-rnd ( a d N --)
    0 swap 0 do
        3dup cells + !
        1+ rot rnd drop -rot
    loop drop drop drop ;
```

15. Lo natural es efectuar un bucle e interrumpirlo en caso de que se encuentre la coincidencia entre el valor suministrado y uno de los elementos del vector. Con `unloop exit` puede romperse el bucle y salirse de la función. Si se da el caso, se deposita un -1 en la pila (verdadero) y se abandonan bucle y función. Si se completa el bucle, se deposita un 0 en la pila (falso). En cualquier caso, se eliminan de la pila la dirección base y el valor buscado, quedando sólo el índice y la bandera. Esta es una variante del ejercicio resuelto anteriormente en el que implementamos la palabra `v-pos`, que presupone que el valor buscado se encuentra efectivamente en el vector.

```
: v-? ( v d N -- f)
    0 swap 0 do
        3dup cells + @
        = if
            nip nip -1 unloop exit
        then
        1+
    loop nip nip 0 ;
```

16.
```
: exch ( d e --) 2dup @ swap @ rot ! swap ! ;
: v-exch ( a b d N --)
    drop tuck swap cells +
    -rot swap cells + exch ;
```

17. El algoritmo de la burbuja es el más sencillo algoritmo posible de ordenación. Toma el primer elemento del vector y lo compara con el segundo y sucesivos, intercambiándolos cada vez que se encuentre un elemento menor que el primero. Al terminar, el primer elemento del vector será el

más pequeño de todos. A continuación se toma el segundo elemento, y se compara con el tercero y sucesivos, haciendo el intercambio cada vez que se encuentre un elemento menor que el segundo. Se repite este procedimiento desde el primer elemento al penúltimo. Al terminar, el vector se encuentra totalmente ordenado. Esencialmente, tenemos pues dos bucles: uno que recorre los elementos desde el primero hasta el penúltimo; otro que, para cada elemento, toma el siguiente y sucesivos hasta el último, hace las comparaciones y si procede los intercambios. A este segundo bucle le llamaremos «interno», y dividiremos nuestro algoritmo en dos palabras: una que ejecute el bucle interno y otra que ejecute el bucle externo. El bucle interno, `v-sort-in`, espera en la pila un índice, la dirección base del vector y el número de elementos. A su vez, esta palabra tiene dos partes bien diferenciadas: establecer la pila en el estado adecuado para ejecutar el bucle y ejecutar propiamente el bucle. En cuanto a la preparación de la pila, esta parte tendrá el efecto `( i d N -- d+i d+i+1 i+i N)` donde se ha de interpretar que `d+i` es la dirección del elemento de índice `i`.

```
: v-sort-in ( i d N --)
    >r over >r
    swap cells + dup cell+
    r> 1+ r> swap
    do
        2dup
        @ swap @
        < if
            2dup exch
        then
        cell+
    loop drop drop ;
```

El bucle externo es inmediato:

```
: v-sort ( d N --)
    0 -rot dup 1- 0 do
        3dup v-sort-in
        rot 1+ -rot
    loop drop drop drop ;
```

Nota: el algoritmo de la burbuja es, como se ha dicho, el más sencillo de los algoritmos de ordenación. Al mismo tiempo, es el más lento, ya

que tiene un coste proporcional al cuadrado del número de elementos. Como en algorítmica no se tiene en cuenta cual es esa constante de proporcionalidad, ocurre a veces que algoritmos teóricamente deficientes son sin embargo prácticos. Éste es uno de esos casos. En nuestra máquina, un portátil Thinkpad con 512MB de RAM y procesador a 1GHz ejecutando gforth sobre SuSe Linux, la ordenación de un vector de 1000 enteros aleatorios lleva un tiempo inapreciable. Un vector de 4000 enteros aleatorios, del orden de un segundo. Un vector de 8000 enteros aleatorios, del orden de tres segundos. Quiere decirse que, a efectos prácticos, puede ser suficiente casi siempre.

18.
```
: v-invert ( d N --)
    over swap 1 - cells +
    begin
        2dup exch
        1 cells - swap
        1 cells + swap
        2dup >=
    until drop drop ;
```

19.
```
: v-acum ( d N -- )
    over @ swap 1 do
        swap cell+ dup @
        rot + swap 2dup ! swap
    loop drop drop ;
```

2.5. Cadenas

Crearemos una cadena como una estructura que contiene dos celdas y la cadena propiamente dicha. La primera celda indica la longitud máxima de la cadena. La segunda, la longitud efectivamente usada. Así pues, la descripción completa de una cadena en la pila serían tres cantidades: longitud, longitud efectiva y dirección del primer carácter. Ahora bien, tres cantidades son muchas, porque no en todos los casos serán usadas las tres y porque cuando haya más de un argumento de cadena (por ejemplo para concatenar cadenas) tendremos seis números en la pila y será engorroso tratar con ellos. Por ese motivo, la sección `does>` deja sólo la dirección del primer carácter de la cadena, y hay palabras especiales para recuperar la longitud máxima, la efectiva, y para establecer una nueva longitud efectiva. Todas las funciones de cadena se identificarán con el carácter `c` en el nombre. En los comentarios de pila, `d` indica dirección, `N` longitud máxima y `n` longitud efectiva.

1. ```
   : cadena create dup , 0 , allot
            does> 2 cells + ;
   : c-l@ ( d -- N)
       2 cells - @ ;
   : c-le@ ( d --n)
       1 cells - @ ;
   : c-le! ( n d --)
       1 cells - ! ;
   : c-le+1 ( d --)
       dup c-le@ 1+ swap c-le! ;
   : c-le-1 ( d --)
       dup c-le@ 1- swap c-le! ;
   ```

2. ```
   : c-type ( d --)
       dup c-le@ 0 do
           @+1 swap emit
       loop drop ;
   ```

3. `accept` es una de las palabras disponibles en todo entorno Forth. Espera en la pila una dirección y el número máximo de caracteres que se leerán. Deja en la pila el número de caracteres efectivamente leídos. Podemos usarla preparándole la pila antes de llamarla y actualizando la longitud efectiva de la cadena después.

   ```
   : c-accept ( d --)
       dup dup c-l@ accept
       swap c-le! ;
   ```

   Redondearemos el ejercicio escribiendo una versión propia de `accept`, que espera en la pila una dirección y el número máximo de caracteres a leer y deja en la pila el número total de caracteres leídos. Una versión sencilla, desde luego:

   ```
   : intro? 13 = ;
   : accept ( d n -- m)
       0 -rot 0 do
           key dup emit
           dup intro? if
               drop drop unloop exit
           else
               over c!
   ```

```
            1+ swap 1+ swap
        then
    loop drop ;
```

4. La operación de copia incluye la actualización de la celda de longitud efectiva de la cadena y la copia propiamente dicha de la cadena.

```
: c-copia ( d d' --)
    2dup over c-le@ cp*
    swap c-le@ swap c-le! ;
```

5. La concatenación es posible si la longitud total de la cadena destino es mayor o igual a la suma de las longitudes efectivas de origen y destino. La palabra `c-concat?` deja una bandera en la pila indicando si la concatenación es posible o no. La palabra `c-concat` copia la cadena origen a partir del último carácter de la cadena destino, y deja una bandera según si la operación pudo realizarse o no. Actualiza también la longitud efectiva de la cadena destino.

```
: c-concat? ( d d' --f)
    dup c-l@ -rot
    c-le@ swap c-le@ + >= ;
: c-concat ( o d -- f)
    2dup c-concat? if
        2dup c-le@ swap c-le@ + -rot
        tuck
        dup c-le@ +
        over c-le@ cp*
        c-le!
        -1
    else
        drop drop 0
    then ;
```

6. Dos cadenas son iguales si sus longitudes efectivas son iguales, y si coinciden en todos y cada uno de sus caracteres. Así que escribiremos en primer lugar una palabra que compara las longitudes efectivas, indicando si son o no son iguales. La comparación de cadenas puede verse como el caso particular de una palabras más general que toma de la pila dos direcciones y un entero y compara ese número de posiciones sucesivas

a partir de las direcciones dadas. Finalmente, la palabra que buscamos puede escribirse combinando la primera, que compara las longitudes efectivas, y la segunda, proporcionándole las direcciones de comienzo y la longitud efectiva.

```
: c-cmp-le ( d d' --)
    c-le@ swap c-le@ = ;
: c-cmp-n ( d d' n --f)
    0 do
        2@+1 <> if
            2drop 0 unloop exit
        then
    loop 2drop -1 ;
: c-cmp0 ( d d' -- f) \ compara en 'crudo'
    dup c-le@ c-cmp-n ;
: c-cmp ( d d' --f)
    2dup c-cmp-le if
        dup c-le@ c-cmp-n
    else
        2drop 0
    then ;
```

7. Para especificar el intervalo que se va a comparar, se pueden dar las posiciones inicial y final, o bien la posición inicial y el número de caracteres a comparar. Usaremos la segunda opción. En realidad, las dos palabras que presentamos no son más que envoltorios convenientes de `c-cmp-n` donde en lugar de dar directamente las direcciones de comienzo se dan las direcciones de comienzo de las cadenas y el desplazamiento (en el primer caso) o desplazamientos (en el segundo) a partir de esos orígenes, junto con el número de caracteres a comparar.

```
: c-cmp-org= ( d d' n m --)
    >r tuck + -rot + r> c-cmp-n ;
```

Una vez hecho esto, es casi gratis comparar el mismo número de caracteres pero a partir de posiciones diferentes en las dos cadenas. Si `a` y `b` son estas posiciones y `n` el número de caracteres a comparar:

```
: c-cmp-org<> ( d d' a b n --f)
    >r >r rot + swap r> + r> c-cmp-n ;
```

8. Si la cadena matriz tiene longitud l_0 y la subcadena tiene longitud l_1, se trata esencialmente de comparar l_1 caracteres, tomando como origen en la subcadena siempre la dirección del primer carácter, y en la cadena posiciones sucesivas desde el primer carácter hasta, como mucho, el carácter $l_0 - l_1$.

```
: c-subc-f ( d d' -- f)
    2dup c-le@ swap c-le@ swap
    tuck - 1+ 0 do
        3dup c-cmp-n if
            drop drop drop
            -1 unloop exit
        else
            rot 1+ -rot
        then
    loop
    drop drop drop 0 ;
```

Escribimos ahora una segunda versión de la palabra anterior, que en lugar de devolver un *flag* devuelve -1 si la subcadena no está contenida en la cadena o la posición dentro de la cadena a partir de la cual se encuentra la subcadena. En la palabra anterior, la dirección base dentro de la cadena se va actualizando dentro del bucle. Si se localizase la subcadena, sólo hay que restar la dirección base actual dentro de la cadena de la dirección de inicio de la cadena para obtener la posición de la subcadena. Por tanto, esta versión se basa en que, antes de nada, se guarda una copia de la dirección de inicio de la cadena. Tanto si la comparación es exitosa en algún momento como si no, el resto consiste esencialmente en construir la pila para la salida.

```
: c-subc-p ( d d' -- f)
    over swap
    2dup c-le@ swap c-le@ swap
    tuck - 1+ 0 do
        3dup c-cmp-n if
            drop drop swap -
            unloop exit
        else
            rot 1+ -rot
        then
```

```
    loop
    drop drop drop drop -1 ;
```

9. Obviamente, son precisas dos palabras, una para comprobar que un carácter sea una letra minúscula y otra para comprobar que sea mayúscula. Aprovechamos la ocasión para escribir otra que informe de si un carácter es o no espacio en blanco:

```
: ascii-min? dup [char] a >= swap [char] z <= and ;
: ascii-max? dup [char] A >= swap [char] Z <= and ;
: c-min ( d --)
    dup c-le@ 0 do
        dup c@ ascii-max? if
            dup c@ 32 + over c!
        then
        1+
    loop ;
: c-may ( d --)
    dup c-le@ 0 do
        dup c@ ascii-min? if
            dup c@ 32 - over c!
        then
        1+
    loop ;
```

10. Este ejercicio es sólo un caso particular que puede implementarse mediante `c-subc-pos` cuando la subcadena tiene longitud 1. No obstante, escribimos una versión específica.

```
: c-char-pos ( d c -- n)
    over swap
    over c-le@ 0 do
        2dup swap c@ = if
            drop swap -
            unloop exit
        else
            swap 1+ swap
        then
    loop drop drop -1 ;
```

11. Eliminar de una cadena un carácter implica dos operaciones: desplazar una posición hacia atrás todos los caracteres que siguen al que se quiere eliminar y modificar la longitud efectiva de la cadena. Tenemos ya una palabra que mueve bloques de memoria y otra para modificar la longitud efectiva, así que basta combinarlas. Si la dirección base de la cadena es d y p la posición del carácter que se desea eliminar, los argumentos para `mov-` son $d+p+1$, $l-p-1$ y 1, donde l es la longitud efectiva de la cadena.

```
: c-x ( p d  --)
    tuck 2dup + 1+ -rot
    c-le@ swap - 1-
    1 mov-
    c-le-1 ;
```

12. Ahora, hay que mover en primer lugar una porción de cadena de longitud $l-p$ a partir de $d+p$ una posición, actualizar la longitud efectiva y finalmente insertar el carácter:

```
: c-i ( c p d --)
    rot >r 2dup 2dup
    + -rot
    c-le@ swap -
    1 mov+
    tuck + r> swap c!
    c-le+1 ;
```

Es fácil comprobar que las dos últimas palabras no son robustas. `c-x` falla si se pretende eliminar el último carácter de la cadena y `c-i` falla si se pretende «insertar» un carácter más allá del último. Esta es una lección clásica de depuración [2]: los errores se suelen esconder en las esquinas. En este caso, en los límites de la cadena. El algoritmo que hemos implementado presupone que vamos a operar en una posición intermedia y falla si no es así. Se podría discutir que «insertar» no tiene sentido referido a una posición más allá del último carácter, pero es claro que debería ser posible eliminar el último carácter.

En estos casos hay **tres** soluciones: o bien repensar el algoritmo para que funcione en cualquier caso, o bien encapsular la palabra que falla

[2] Véase por ejemplo *La práctica de la programación* de B. Kernigham y R. Pike

en otra que haga las comprobaciones, o bien concluir en que el caso que falla es en realidad una operación de naturaleza distinta (es el caso entre *insertar* un carácter en una cadena y *añadir* un carácter a una cadena). A priori no es fácil elegir qué camino seguir, puesto que un algoritmo que funcione en cualquier caso a) puede ser difícil de encontrar, b) puede que tenga una complejidad que no compense y c) puede que obligue a modificar palabras que no convenga modificar porque sean palabras genéricas que funcionan ya bien para lo que fueron escritas. Pero, desde luego, cuando este algoritmo existe y es adecuado ha de preferirse.

En este caso, la decisión es: respecto a `c-x` la encapsularemos en una `c-X` que comprueba si la posición que se quiere eliminar es la última de la cadena, en cuyo caso se limita a decrementar en una unidad la longitud efectiva de la cadena; respecto a `c-i`, concluimos que la inserción de un carácter es una operación de naturaleza distinta al añadido de un carácter más allá del último, por lo que implementaremos una palabra `c-a`

```
: c-X ( p d --)
    2dup c-le@ 1- < if
        c-x
    else
        nip c-le-1
    then ;

: c-a ( c d --)
    tuck dup c-le@ + c!
    c-le+1 ;
```

13. Se trata de encontrar la posición del primer carácter dentro de la cadena que es distinto del espacio en blanco. Si p es esta posición, se trata únicamente de desplazar p posiciones a la izquierda un bloque de $l-p$ bytes, siendo l la longitud efectiva de la cadena, y actualizar dicha longitud efectiva a $l-p$. La palabra `c-1` devuelve la posición del primer carácter distinto de espacio en blanco (ascii 32). Presuponemos que la cadena no está vacía.

```
: c-1 ( d --p) 0 begin 2dup + c@ 32 = while 1+ repeat nip ;
: c-no-blancos< ( d --)
    dup c-1
```

```
    2dup 2dup
    + -rot swap c-le@ swap -
    rot dup >r mov- r>
    over c-le@ swap -
    swap c-le! ;
```

14. Se trata de encontrar el último carácter dentro de la cadena distinto del espacio en blanco, y ajustar la longitud efectiva. Si la posición de este último carácter es p la nueva longitud efectiva es $l-p$. La palabra `c-$` devuelve la posición del último carácter distinto de espacio. De nuevo, presuponemos que la cadena no está vacía.

    ```
    : c-$ ( d --p)
        dup c-le@  1-
        begin 2dup + c@ 32 = while 1- repeat nip ;
    : c-no-blancos> ( d--)
        dup c-$ 1+ swap c-le! ;
    ```

    Puede ser interesante conocer la longitud de una cadena desde el primer carácter distinto de espacio hasta el último carácter distinto de espacio. Podemos llamar a esta longitud *longitud aparente*, ya que los espacios no son visibles. Es fácil de implementar:

    ```
    : c-la ( d --n)
        dup c-$ swap c-1 - 1+ ;
    ```

15. El procedimiento que seguiremos será el siguiente: tomaremos la dirección de comienzo de la cadena y la dirección siguiente, e iremos incrementando ambas direcciones hasta que la segunda alcance el último carácter de la cadena. Cuando en la primera dirección se encuentre un espacio en blanco y en la segunda un carácter distinto del espacio, entonces esta segunda dirección apunta al principio de una palabra. Entonces, se guarda en la pila la dirección y se incrementa un contador, también en la pila, que indica el número de direcciones apiladas hasta el momento. De esta forma, la estructura de la pila es: una o varias direcciones, un número que indica el número de esas direcciones, un puntero a un lugar de la cadena y un puntero a la posición siguiente. Al final, quedan en la pila una serie de direcciones y su número. Esos valores pueden trasladarse a un vector mediante `v!!`. Es preciso considerar el caso en que el primer carácter de la cadena sea distinto de espacio, en cuyo caso es comienzo de palabra.

```
: c-palabras ( d -- a b c ... n)
    dup c@ 32 <> if
        1 over
    else
        0 swap
    then
    dup 1+
    over c-le@ 1- 0 do
        2@+1
        32 <> swap 32 = and
        if >r swap 1+ over r> then
    loop drop drop ;
```

Por otro lado, el colocar todas las direcciones en la pila puede ser indeseable en según qué situaciones, pues otros elementos de la misma quedarían inaccesibles. Sería entonces útil otra versión que, dada una dirección, obtuviese la dirección de la palabra siguiente. Un problema secundario es que si se desea explorar la cadena en su totalidad, puesto que no hay marca de final ¿dónde detener la búsqueda? Escribiremos dos palabras nuevas: `c-#p` proporciona el número de palabras en una cadena; `c-ps` dada una dirección ofrece la dirección de la palabra siguiente. De esta forma se puede usar la primera para hacer un bucle en cuyo interior se irá pasando sucesivamente de una palabra a la siguiente. Esta implementación de `c-ps` que sólo precisa una dirección y devuelve una dirección falla si el primero o el último carácter de la cadena son distintos de espacio en blanco. Pero preferimos el uso limpio de la pila que hace, así que, cuando sea preciso analizar una cadena, tomaremos la precaución de insertar al principio y añadir al final un espacio en blanco. Para cuando esto sea preciso, escribimos `c-bordes`

```
: c-#p ( d --n)
    dup c@ 32 = if
        0 swap
    else
        1 swap
    then                            \ c d
    dup c-le@ 1- >r
    dup 1+ r> 0 do                  \ c d d+1
        2@+1 32 <> swap 32 = and if
            rot 1+ -rot
        then
```

```
    loop 2drop ;
: c-saltar-blancos ( d --d')
    begin
        dup c@ 32 =
    while
        1+
    repeat ;
: c-saltar-no-blancos ( d --d')
    begin
        dup c@ 32 <>
    while
        1+
    repeat ;
: c-bordes ( d -- d)
    dup 32 swap 0 swap c-i
    dup 32 swap c-a ;
: c-ps ( d --d')
    saltar-no-blancos
    saltar-blancos ;
```

16.
```
: c-vacia? ( d --f)
    dup c-le@ 0 do
        @+1 32 <> if
            drop 0 unloop exit
        then
    loop drop -1 ;
```

17. La idea aquí es tomar la dirección base de la cadena y la dirección siguiente, e incrementarlas hasta que la segunda alcance el final de la cadena. Si los contenidos de ambas direcciones son espacio en blanco, entonces se elimina el carácter apuntado por la segunda mediante `c-x`. Puesto que la longitud efectiva de la cadena va cambiando a medida que se eliminan caracteres, no podemos recorrerla mediante un bucle `do...loop`.

```
: c-purgar ( d --)
    0 begin
        2dup swap c-le@ 2 - <=
    while
        2dup + dup 1+
        c@ 32 = swap
```

```
        c@ 32 = and if
            2dup 1+  swap c-x
        else
            1+
        then
    repeat 2drop ;
```

18. El problema aquí es que es preciso controlar que no se explora más allá del final de la cadena. Por eso es útil una palabra que, dados un índice y una dirección base, indique si el índice apunta a alguna posición dentro de la cadena.

```
: c-dentro? ( n d --f)
    c-le@ < ;
: c-lpn ( n d --m)
    0 -rot begin
        2dup c-dentro? >r
        2dup + c@ 32 <> r> and
    while
        rot 1 -rot
        swap 1+ swap
    repeat drop drop ;
```

Esta versión no es adecuada para usar junto con `c-palabras`, pues ésta deja en la pila direcciones absolutas, y no parejas (desplazamiento, dirección base). Por ese motivo escribiremos otra versión que toma sólo una dirección y devuelve el número de caracteres distintos del espacio en blanco seguidos a partir de esa dirección. Un problema es que la cadena carece de marca de final, y por tanto la palabra fallará si el último carácter de la cadena es distinto de espacio en blanco.

: c-lp (d –n) 0 begin over c@ 32 ¡¿while 1+ swap 1+ swap repeat nip ;

19. Ajustemos a la izquierda. La palabra `c-1` indica el número de espacios en blanco que es preciso eliminar. Puesto que ha de conservarse la longitud efectiva de la cadena, por cada espacio eliminado al principio de la cadena es preciso añadir otro al final de la misma.

```
: c-< ( d --)
    dup c-1 0 do
        dup 0 swap c-x
        dup 32 swap 0 swap c-i
    loop drop ;
```

Para ajustar a la derecha es preciso eliminar espacios en blanco al final e introducirlos al principio.

```
: c-> ( d --)
    dup dup c-le@ swap c-$ - 1- ?do
        dup dup c-le@ + 1- c-x
        dup 32 swap 0 swap c-i
    loop drop ;
```

Ajustamos a la derecha:

```
: c-> ( d --)
    dup dup c-le@ swap c-$ - 1- 0 ?do
        dup dup c-le@ 1- swap c-x
        dup 32 swap 0 swap c-i
    loop drop ;
```

Para ajustar al centro, es útil el concepto de longitud aparente, que hemos definido e implementado mediante `c-la`:

```
: c->< ( d --)
    dup c-<
    dup dup c-le@ swap c-la
    - 2 / 0 ?do
        dup dup c-le@ 1- swap c-x
        dup 32 swap 0 swap c-i
    loop drop ;
```

Estas palabras son útiles para imprimir en columnas. Dada una cadena, podemos suplementarla con espacios en blanco hasta que alcance una longitud determinada, y luego ajustarla a izquierda, derecha o centro antes de imprimirla.

2.6. Matrices

1. Aparte de conocer el número de filas y columnas, es interesante conocer el número total de elementos de la matriz, ya que, puesto que sus elementos se disponen linealmente en la memoria, habrá situaciones en que resulte conveniente considerarla como un vector (p. ej. para encontrar

los valores máximo y mínimo y otros similares). Por eso introducimos `m-N` que toma una dirección y devuelve esa misma dirección y el número de elementos de la matriz, quedando entonces la pila apta para usar las funciones de vectores ya desarrolladas.

```
: matriz create 2dup swap , , * cells allot
       does> 2 cells + ;

: m-f ( d --f)
    2 cells - @ ;
: m-c ( d --f)
    1 cells - @ ;
: m-N ( d -- d N)
    dup m-f over m-c * ;
```

2. Habilitamos un mecanismo, si bien sencillo, para evitar las lecturas o escrituras fuera de las dimensiones de la matriz. Si f es el número de filas y c el número de columnas de la matriz, el desplazamiento en celdas desde la dirección de comienzo del elemento de coordenadas (i, j) es $i \times c + j$. Este desplazamiento ha de ser menor que $f \times c$. Las filas y columnas se indexan desde 0. Ya que dados un par de valores de fila y columna y la dirección base de la matriz necesitaremos en más de una ocasión encontrar la dirección del elemento dado, factorizaremos esta operación en una palabra `m-dir` que se ocupa además de que no se vaya más allá de los límites.

```
: m-ij->l ( i j d -- d')
    rot over m-c * rot +
    over dup m-c swap m-f * min
    cells + ;

: m! ( v i j d --)
    m-ij->l ! ;
```

3.
```
: m@ ( i j d -- v)
    m-ij->l @ ;
```

4. Separamos la impresión en dos palabras. El bucle interno recorre los elementos de una fila e imprime el `cr` antes de pasar a la siguiente línea

```
: m.0 ( i d --)
    0 swap dup m-c 0 do
        3dup m@ 12 .r swap 1+ swap
    loop cr drop drop drop ;
: m. ( d --)
    cr 0 swap dup m-f 0 do
        2dup m.0
        swap 1+ swap
    loop drop drop ;
```

5. Esta es una situación en que podemos considerar a la matriz como un vector. Tenemos ya la función equivalente para vectores, luego sólo nos resta obtener el número de elementos de la matriz.

```
: 0m!! ( d --)
    dup m-c over m-f * 0v!! ;
```

6. Al igual que en la palabra anterior, usamos la versión ya preparada para vectores.

```
: m-map ( xt d -- )
    dup m-c over m-f * v-map ;
```

7. Este caso es ligeramente distinto al anterior y no podemos hacer una llamada a `v-map-i` desde la nuestra ya que en la versión para vectores `execute` espera un argumento en la pila y ahora necesitamos dos. Sí podemos retener el punto de vista de que la matriz es un vector, y para cada índice obtener la fila y columnas correspondientes, aplicando la función sobre ellas.

```
: m-l->ij ( l d -- i j)
    m-c /mod swap ;
: m-map-ij ( xt d --)
    0 swap dup m-c over m-f * 0 do
        3dup 2dup swap cells + >r
        m-l->ij rot execute
        r> !
        swap 1+ swap
    loop drop drop drop ;
```

8. Aprovechamos la versión para vectores.

```
: m-max ( d -- )
    m-N v-max ;
: m-min ( d -- )
    m-N v-min ;
```

9. De nuevo, aprovechamos la versión para vectores:

```
: m-rnd ( s d --)
    dup m-c over m-f * v-rnd ;
```

   Es obvio que muchas otras funciones para matrices pueden basarse en las funciones para vectores, y no hay que insistir en ello ya que el procedimiento es obvio.

10. Para esta palabra como para la siguiente será útil una palabra que intercambie dos elementos cualesquiera de una matriz. Esperará en la pila fila y columna del primer elemento, fila y columna del segundo elemento y dirección base:

```
: m-exch ( i j p q d --)
    dup >r m-ij->l -rot r> m-ij->l exch ;
```

   Ahora es fácil intercambiar dos filas:

```
: m-exch-f ( i j d --)
    0 swap dup m-c 0 do
        2over 2over
        >r tuck r> m-exch
        swap 1+ swap
    loop
    4 0 do drop loop ;
```

   o dos columnas

```
: m-exch-c ( i j d --)
    0 swap dup m-f 0 do
        2over 2over
        >r dup >r -rot r> swap r> m-exch
        swap 1+ swap
    loop
    4 0 do drop loop ;
```

11. Para trasponer una matriz es preciso recorrer todas sus filas, menos la última, y para cada fila i considerar los elementos que se encuentran en las columnas $j > i$. Por eso factorizamos la operación en dos palabras, una para cada bucle:

```
: m-'-f ( i d --) \ para la fila i
    over 1+ tuck over m-c swap do
        3dup >r 2dup swap r>
        m-exch
        swap 1+ swap
    loop drop drop drop ;
: m-' ( d --)
    0 swap dup m-f 1- 0 do
        2dup m-'-f
        swap 1+ swap
    loop drop drop ;
```

12. Multiplicación de una fila o columna por un número. Se esperan en la pila el número, la fila o columna y la dirección base de la matriz.

```
: m-f* ( n i d --)
    0 swap dup m-c 0 do
        2over 2over
        m-ij->l dup >r @ * r> !
        swap 1+ swap
    loop
    4 0 do drop loop ;

: m-c* ( n j d --)
    0 -rot dup m-f 0 do
        2over 2over
        m-ij->l dup >r @ * r> !
        rot 1+ -rot
    loop
    4 0 do drop loop ;
```

13. La idea aquí es posicionarse en las direcciones de comienzo de cada fila y avanzar una celda cada vez tantas veces como columnas tenga la matriz. Las palabras esperan en la pila las dos filas y la dirección base de la matriz. La primera fila es sustituida por su suma, o resta, con la segunda, que queda inalterada.

```
: m-ff'+ ( f f' d --)
    dup m-c >r
    tuck 0 swap m-ij->l -rot 0 swap m-ij->l
    r> 0 do
        2@+c + over 1 cells - !
    loop 2drop ;
: m-ff'- ( f f' d --)
    dup m-c >r
    tuck 0 swap m-ij->l -rot 0 swap m-ij->l
    r> 0 do
        2@+c swap - over 1 cells - !
    loop 2drop ;
```

2.7. Cálculo numérico

En esta sección se suponen operaciones con números reales. Las palabras para manipular este tipo de números se encuentran en el manual de Gforth, que es el sistema que estamos usando, a partir de la página 60 en la edición que estamos manejando [3]. En general, existe un paralelismo entre las palabras de enteros y las palabras de reales, p. ej. `drop` para enteros y `fdrop` para reales. Una diferencia importante es que el tamaño de los reales no coincide con el de los enteros. La palabra equivalente a `cells` es `floats`. Otra, que los reales se tratan en su propia pila, y que no existe `f>r`. Para retener la funcionalidad de `>r` podemos colocar el real en algún lugar «seguro», como `here`. Por ejemplo:

```
: f>r here f! ;
: fr> here f@ ;
```

Podríamos igualmente implementar una pila propia para reales, pero nos contentamos con esta solución sencilla. Hay que tener en cuenta además que cuando en la «pila» se encuentren argumentos reales y enteros, como es el caso del primer ejercicio, tratamos en realidad con dos pilas. Para evitar confusiones, separamos los contenidos de ambas en los comentarios mediante dos puntos. Añadiremos a continuación las contrapartidas para reales que nos vayan siendo útiles:

```
: f2dup fover fover ;
```

[3] Al ser un documento en continuo proceso de revisión no podemos garantizar la coincidencia con otras ediciones o revisiones

```
: -frot frot frot ;
: s>f s>d d>f ;
: f2over ( x y z t -- x y z t x y)
    here f!
    here 1 floats + f!
    f2dup
    here 1 floats + f@ -frot
    here f@ -frot ;
: @+f ( : d -- v : d+float)
    dup f@ 1 floats + ;
```

1.
```
: espacio 32 emit ;
: espacios 0 do espacio loop ;

: ftabular ( h a b : xt --)
    cr
    begin
        f2dup f<
    while
        fover f. 10 espacios
        fover dup execute f. cr
        -frot fover f+
        frot
    repeat
    drop                \ ahora tengo que limpiar
    3 0 do fdrop loop ; \ dos pilas
```

2.
```
: fvector create dup , floats allot
        does> dup @ swap cell+ swap ;

: fv! ( v i d N --)
    rot min floats + f! ;
: fv@ ( i d N --)
    rot min floats + f@ ;
```

3.
```
: ftab ( a h : xt d N --)
    0 do
        fover over execute   \ a h v : xt d
        dup f!               \ a h : xt d
        ftuck f+ fswap       \ a+h h : xt d
        1 floats +           \ a+h h : xt d'
```

```
    loop drop drop fdrop fdrop ;
```

Podemos preparar una tabla de argumentos igualmente espaciados para aplicarles alguna función. Si declaramos la función identidad:

```
: f-identidad ;
```

usada en combinación con ftab nos permite rellenar un vector con argumentos. Por ejemplo, para rellenar un vector de 10 elementos `X` desde 0,0e hasta 0,9e a intervalor de una décima: `0.0e 0.1e ' f-identidad X ftab`. En este contexto, es también útil la versión para reales de `v-map`, que aplica una función a cada elemento del vector:

```
: fv-map ( : xt d N --)
    0 do
        2dup f@ execute
        dup f!
        1 floats +
    loop drop drop ;
```

Puede ser útil disponer por separado de dos vectores: el vector de argumentos y el vector de valores obtenidos a partir de ellos por aplicación de una función dada. Para eso, convendrá una palabra para copiar un vector de reales en otro:

```
: fv-copia ( : d N d' N --)
    drop swap floats 1 cells + cp* ;
```

4. Esta es la versión para reales de `v!!`

```
: fv!! ( a1...aj m d N --)
    rot min 0 do
        dup f! 1 floats +
    loop drop ;
```

El único inconveniente es que los valores han de introducirse en orden inverso. No es problema porque disponemos de la palabra `v-invert` de la que podemos escribir una versión para vectores de reales:

```
: fexch ( d d' --)
    2dup f@ f@ f! f! ;
: fv-invert ( d N --)
    over swap 1 - floats +
    begin
        2dup fexch
        1 floats - swap
        1 floats + swap
        2dup >=
    until drop drop ;
```

5. Hay que observar que la serie es una suma de término general $T_n = x^n/n!$ y que existe la relación de recurrencia $T_{n+1} = xT_n/(n+1)$. Aprovecharemos esta circunstancia para agilizar el cálculo. Por otra parte, la serie es infinita y habremos de calcular hasta un número máximo de términos. El criterio que adoptaremos será cesar el cálculo en el término $T_n < 10^{-9}$. La pila contendrá n que se incrementará en 1 en cada término. Respecto a la pila de reales, contendrá la suma parcial, el argumento y el último término.

```
: fe^ ( x -- e^x)
    1                \ n
    1.0e fswap       \ suma parcial
    1.0e             \ T0
    begin
        fdup 1e-9 f>
    while
        fover f*
        dup s>f f/  1+
        frot fover f+ -frot
    repeat
    fdrop fdrop drop ;
```

6. En el método de la bisección, se toma el valor intermedio $m = (a+b)/2$ y se comprueba si el signo de $f(m)$ es igual o distinto a signo de $f(b)$. Si es distinto, la raíz se busca en el intervalo $[m, b]$. Si es igual, la raíz se busca en el intervalo $[a, m]$. Consideramos haber encontrado la raíz r si $f(r) < 10^{-9}$. De la propia definición del algoritmo se sigue una implementación recursiva. La palabra que vamos a escribir espera en la pila de reales los extremos del intervalo y en la pila la dirección de la función cuya raíz se busca. Presuponemos que existe dicha raíz.

```
: fbisec ( a b : xt -- r:)
    fover dup execute               \ es 'a' la raiz?
    1e-9 f< if fdrop drop exit then
    fdup dup execute                \ es 'b' la raiz?
    1e-9 f< if fnip drop exit then

    f2dup f+ 2.0e f/                \ punto medio 'm'
    dup execute fswap
    dup execute f*                  \ f(b)*f(m)
    f0< if
        frot fdrop fswap recurse    \ busca en [m,b]
    else
        fnip recurse                \ busca en [a,m]
    then ;
```

7. La idea aquí es sencilla: dividir el intervalo en subintervalos, comprobar para cada subintervalo si hay un cambio de signo en la función y, si es así, encontrar la raíz en ese subintervalo mediante la palabra `fbisec`.

```
: hay-raiz? ( a b : xt -- :f)
    dup execute fswap execute
    f* f0<= ;
: fraices ( a b : xt --)
    cr fover f- 1e2 f/              \ a h : xt
    100 0 do
        f2dup fover f+
        dup hay-raiz? if
            f2dup fover f+ dup
            fbisec f. cr
        then
        fswap fover f+ fswap
    loop fdrop fdrop drop ;
```

8. Necesitamos en primer lugar una raíz aproximada. Una ver obtenida ésta calculamos sucesivas aproximaciones hasta que la diferencia entre dos de ellas sucesivas sea menor de 10^{-9}. `fsqrt0` calcula una raíz aproximada de su argumento. `fsqrt1` calcula una raíz mejor a partir del argumento y una raíz anterior.

```
: fsqrt0 ( x: -- r:)
```

```
    1.0e begin
        f2dup fdup f* f>
    while
        1.0e f+
    repeat fnip ;
: fsqrt1 ( x r0: -- x r0 r1:)
    f2dup f/ fover f+ 0.5e f* ;

: fsqrtN ( x: -- r:)
    fdup fsqrt0 fsqrt1
    begin
        f2dup f- fabs 1.0e-9 f>
    while
        fnip fsqrt1
    repeat fnip fnip ;
```

9. La palabra que escribiremos tiene dos partes, y podría ser factorizada. En la primera parte, se rellena el vector con los valores de la función: $f(a)$, $f(a+h)$... hasta $f(b)$. En la segunda se calcula la integral. En la pila se esperan la dirección de la función que se desea integrar, la dirección del vector y su número de elementos. En la pila de reales a y b. Si el vector tiene n elementos, entonces el intervalo h es $(b-a)/(n-1)$. No es necesario declarar un vector a propósito: puede usarse espacio a partir de `here`

```
: fintegral ( a b: xt d N -- i:)

    2dup >r >r rot r> r>         \ guardo (d,N)
    fover f- dup 1- s>f f/       \ calculo h
    0 do                         \ relleno el vector con una
        fover over execute       \ tabla de valores
        dup f! 1 floats +
        ftuck f+ fswap
    loop fnip drop drop          \ conservo 'h'

    0.0e                         \ acumulador s
    1- 0 do                      \ s:d
        dup f@
        1 floats + dup f@
        f+ 0.5e f* f+
    loop f* drop ;
```

10. Dado el conjunto de valores (x_i, y_i), pretendemos ajustarlos mediante una recta $y = ax+b$. El ajuste óptimo se consigue para aquellos valores (a, b) tales que la separación entre los puntos y la recta es mínima. Esto se formaliza considerando la función

$$S(a, b) = \sum_i [y_i - (ax_i + b)]^2 \tag{2.3}$$

que será mínima cuando sus dos derivadas parciales, respecto a a y respecto a b sean nulas. De aquí se sigue un sistema de dos ecuaciones que proporciona (a, b):

$$\begin{aligned} as_{x^2} + bs_x &= s_{xy} \\ as_x + bN &= s_y \end{aligned} \tag{2.4}$$

o

$$a = \frac{Ns_{xy} - s_x s_y}{Ns_{x^2} - (s_x)^2} \tag{2.5}$$

$$b = \frac{s_{x^2} s_y - s_x s_{xy}}{Ns_{x^2} - (s_x)^2} \tag{2.6}$$

donde

$$\begin{aligned} s_x &= \sum_i x_i \\ s_{x^2} &= \sum_i x_i^2 \\ s_y &= \sum_i y_i \\ s_{xy} &= \sum_i x_i y_i \end{aligned} \tag{2.7}$$

Así pues, de entrada, hemos de escribir palabras que, dado un vector en pila, o ambos, calculen las sumatorias:

```
: fv-sum ( :d N -- s:)
    0.0e 0 do
```

```
        dup f@ f+
        1 floats +
    loop drop ;
: fv-sum2 ( :d N -- s2: )
    0.0e 0 do
        dup f@ fdup f* f+
        1 floats +
    loop drop ;
: fv-sumxy ( : d N d' N -- sxy:)
    0.0e drop swap 0 do
        2dup f@ f@ f* f+
        1 floats + swap 1 floats +
    loop drop drop ;
```

y dado que en las expresiones para a y b el denominador es común (es el determinante de la matriz de coeficientes), escribimos también una palabra para calcularlo expresamente:

```
: fv-det ( :d N -- v:)
    2dup fv-sum2 dup s>f f*
    fv-sum fdup f* f- ;
```

Ya podemos calcular la pendiente, suponiendo que en la pila se encuentran depositados primero el vector para las x y a continuación el vector para las y.

```
: fv-pendiente ( : d N d' N -- a:)
    2over fv-det            \ determinante
    dup s>f                 \ N
    2over 2over fv-sumxy    \ sum x.y
    fv-sum                  \ sum y
    fv-sum                  \ sum x
    f* -frot f* fswap f-
    fswap f/ ;

: fv-ordenada ( : d N d' N -- b:)
    2over fv-det            \ determinante
    2over 2over fv-sumxy    \ sum x.y
    fv-sum                  \ sum y
    f2dup fv-sum            \ sum x
```

```
    fv-sum2                 \ sum x.x
    frot f* -frot f* f-
    fswap f/ ;

: fv-ajuste-lineal ( : d N d' N -- a b:)
    2over 2over
    fv-pendiente
    fv-ordenada ;
```

11. Acudiendo a la definición de derivada, el problema es trivial:

```
: f' ( x : xt -- y:)
    1e-9 fswap f2dup f+
    dup execute fswap execute f-
    fswap f/ ;
```

12.
```
: fpol-eval ( x: d N -- y:)
    0e fswap
    0 swap 0 do
        2dup fdup
        s>f f** f@ f*
        frot f+ fswap
        1+ swap 1 floats + swap
    loop drop drop fdrop ;
```

    Si se introducen los coeficientes del polinomio mediante `fv!!`, téngase presente que los valores se almacenan en orden inverso.

13. Un polinomio de coeficientes $a_0, a_1, a_2, a_3, ..., a_n$ tiene una derivada que se representa por el polinomio $a_1, 2a_2, 3a_3, ..., 0$

```
: fpol-derivada ( : d N --)
    1 swap 1- 0 do
        2dup s>f 1 floats + f@ f*
        over f!
        1+ swap 1 floats + swap
    loop drop 0.0e f! ;
```

14. Establecemos a 0 la constante de integración

```
: fpol-integral ( : d N --)
    2dup 1- floats +
    rot drop swap 1- dup
    0 do
        2dup s>f 1 floats - f@ fswap f/
        over f!
        1- swap 1 floats - swap
    loop drop 0.0e f! ;
```

15. Sirviéndonos de la palabra anterior, es fácil encontrar la integral definida de un polinomio en un intervalo, teniendo en cuenta que si D es una primitiva de $p(x)$ entonces

$$\int_a^b p(x)dx = D(b) - D(a) \tag{2.8}$$

```
: fpol-integral-definida ( a b:d N--x:)
    2dup fpol-integral
    2dup fpol-eval fswap fpol-eval
    f- ;
```

16. Para la ecuación propuesta, si h es el incremento de tiempo, $x(t+h)$ se calcula a partir de $x(t)$ a través de los siguientes pasos:

    a) $k_1 = f(t)$
    b) $k_2 = f(t + \frac{h}{2})$
    c) $k_3 = f(t + \frac{h}{2})$
    d) $k_4 = f(t + h)$
    e) $x(t+h) = x(t) + \frac{h}{6}[k_1 + 2(k_2 + k_3) + k_4]$

    Escribiremos una palabra que espere en las pilas (x_0, t_0, h) por un lado y por otro la dirección de la función $f(t)$ y dos vectores para almacenar los valores sucesivos de t y de x. En primer lugar, usando `ftab`, rellenamos el vector que contiene los valores para el tiempo. Después a partir del valor inicial de x obtenemos los sucesivos y los vamos almacenando.

```
: frk4-ft ( x0 t0 h : xt d N d' N --)

    f2dup 2swap ['] f-identidad
```

```
    -rot ftab                       \ vector con valores de t

    0 do
        frot fdup dup f! -frot
        1 floats +                  \ guardo x
        ftuck                       \ guardo h
        fover over execute -frot    \ k1
        f2dup 0.5e f* f+ over
        execute -frot               \ k2
        f2dup f+ over
        execute -frot               \ k4
        f+ f>r
        fswap 4.0e f* f+ f+
        fover 6.0e f/ f*
        frot f+ fswap
        fr> fswap
    loop fdrop fdrop fdrop drop drop ;
```

17. Para la ecuación propuesta, si h es el incremento de tiempo, $x(t+h)$ se calcula a partir de $x(t)$ a través de los siguientes pasos:

    a) $k_1 = f(x)$
    b) $k_2 = f(x + \frac{k_1}{2})$
    c) $k_3 = f(x + \frac{k_2}{2})$
    d) $k_4 = f(x + k_3)$
    e) $x(t+h) = x(t) + \frac{h}{6}\left[k_1 + 2(k_2 + k_3) + k_4\right]$

    Como se ve, los cálculos intermedios no precisan de t

```
: frk4-fx ( x0 t0 h : xt d N d' N --)

    f2dup 2swap ['] f-identidad
    -rot ftab                       \ vector con valores de t

    fnip fswap 0 do                 \ h x : xt d
        fdup dup f! 1 floats +
        fdup over execute fswap     \ k1
        f2dup fswap 0.5e f* f+
        over execute fswap          \ k2
```

```
        f2dup fswap 0.5e f* f+
        over execute fswap          \ k3
        f2dup f+
        over execute fswap          \ k4

        f>r -frot f+ 2.0e f*
        f+ f+ fover 6.0e f/ f* fr> f+
    loop fdrop fdrop drop drop ;
```

Aunque el código anterior es correcto, los resultados son inaceptables. Por ejemplo, cuando $f(x) = 1/x$ la solución de la ecuación cuando $x(0) = 1$ es $x = \sqrt{2(t + \frac{1}{2})}$ que para $t = 0{,}99$ es $x(0{,}99) = 1{,}726267$ mientras que numéricamente obtenemos 1,613285. Esto ilustra los peligros del cálculo numérico con números reales. En este caso, conocemos la solución analítica, y por tanto hemos podido detectar el error. ¿qué ocurre cuando integramos una ecuación de la que desconocemos la solución analítica (por eso precisamente la integramos)? El mismo algoritmo implementado en C y compilado mediante gcc da resultados correctos, tanto con tipo *float* como con tipo *double*.

18. Necesitamos un generador de números aleatorios. Ya disponemos de un generador de enteros, a partir del cual podemos escribir otro de reales. Una vez hecho esto, la implementación es sencilla

```
: frnd
    rnd
    s>f 32768 s>f f/ ;

: fmontecarlo2 ( :xt -- v:)
    0.0e rnd-seed
    1000000 0 do
        frnd frnd over execute f+
    loop 1e6 f/ drop ;
```

2.8. Archivos

1. Las funciones básicas son `open-file` y `read-line`. La primera espera en la pila la dirección que contiene el nombre del archivo, la longitud de ese nombre y el modo en que será abierto. Devuelve un entero que identifica al archivo para posteriores operaciones y un *flag* a cero. Si

hubo algún problema, en lugar del identificador del archivo devuelve 0 y en lugar del *flag* un código de error. En cuanto a `read-line` espera en la pila la dirección y tamaño del buffer en que será guardada la línea leída y el identificador del archivo. Devuelve el número de caracteres leídos y un *flag*: -1 si todo fue bien y 0 en caso contrario. También deja un valor 0 en la pila. Con esto, la impresión por pantalla de un archivo requiere la apertura del archivo y la lectura de líneas en tanto la operación sea posible. La implementación es sencilla, y aquí suponemos que no hay condiciones de error (como que el archivo no exista o no pueda abrirse). Esta palabra pues tendría que ser encapsulada en otra que considerase estas condiciones de error. `list-file` espera en la pila las direcciones de dos cadenas (del formato que definimos en la sección dedicada a las cadenas): la que contiene el nombre del archivo y la del buffer donde serán depositadas las líneas.

```
: f-list ( d d' --)
    swap c-le@ r/o open-file drop     \ d' id
    begin
        2dup swap dup c-l@ rot
        read-line drop                \ d' id n flag
    while
        >r over r> swap c-le!
        over c-type cr
    repeat
    swap close-file 2drop ;
```

2. Esta es una ligera variante

```
: f-list# ( d d' --)
    cr
    swap dup c-le@ r/o
    open-file drop                    \ d' id
    1 -rot                            \ cuenta lineas
    begin
        2dup swap dup c-l@ rot
        read-line drop                \ c d' id n flag
    while
        >r over r> swap c-le!         \ c d' id
        rot dup 6 .r 1+ -rot
        2 espacios over c-type cr
```

```
    repeat
    swap close-file 3drop ;
```

3. Esta palabra tomará como argumentos el nombre del archivo y el buffer donde se leerán las cadenas. De nuevo, no hay gestión de errores.

```
: f-lee-lineas ( d d' --)
    cr s" C-n nueva linea; C-f fin"  type cr
    swap dup c-le@ w/o open-file drop  \ d' id
    begin
        key 14 =
    while
        over c-accept
        2dup write-line drop
    repeat 2drop ;
```

4. Esta palabra necesita tres argumentos: el nombre del archivo, el nombre del buffer donde serán leídas las líneas y el vector donde los números serán almacenados. Vamos a suponer que no hay líneas en blanco y que el número de líneas del archivo coindice con el número de líneas del vector. Una implementación robusta comprobaría que esto es así, pero ésta es sólo un ejercicio. Necesitamos una palabra capaz de convertir una cadena que representa un entero en un entero, con su signo.

```
: c-a-entero ( d n -- v)
    over c@ [char] - = if
        -1 -rot
        1- swap 1+ swap
    else
        1 -rot
    then
    0 -rot
    0 do
        @+1 48 -
        rot 10 * + swap
    loop drop * ;

: f-a-vector ( d d' d'' N --)  \ archivo, buffer y vector
    >r rot dup c-le@ r/o
    open-file drop             \ d' d'' id
```

```
r> 0 do                       \ d' d'' id
    rot 2dup                  \ d'' id d' id d'
    dup c-l@                  \ d'' id d' id d' l
    rot read-line             \ d'' id d' n flag 0
    drop drop                 \ d'' id d' n
    over c-le!                \ d'' id d'
    dup c-<
    dup dup 0 swap            \ d'' id d' d' 0 d'
    c-lpn                     \ d'' id d' d' l
    c-a-entero                \ d'' id d' v
    >r rot r> over !          \ id d' d''
    1 cells + rot
loop close-file 2drop ;
```

Aunque es una palabra algo larga, se mantiene simple gracias que expresa sólo una secuencia lineal de operaciones. En primer lugar se prepara la pila para leer una línea, que es depositada en el buffer. Se eliminan mediante `c-<` los posible espacios en blanco al principio y se calcula la longitud de la palabra que representa al número. Esto es importante: no podemos usar la longitud efectiva de la cadena puesto que ésta podría incluir caracteres en blanco adicionales tras los dígitos. Una vez hecho esto, ya puede calcularse el entero que representa y almacenarse en el vector. Después de incrementar la dirección del vector en una celda y dejar la pila tal y como estaba al inicio del bucle, se vuelve al principio y se lee la línea siguiente. Acabado el bucle, cerramos el archivo y limpiamos la pila.

5. Es natural usar la palabra `c-palabras` que ya tenemos. Esta palabra espera en la pila la dirección de una cadena, y devuelve las direcciones de las palabras contenidas en la cadena y el número total de palabras, supuesto que la cadena no se encuentra vacía (existe otra palabra que hemos escrito para comprobar esto). Ahora bien: si la palabra que escribamos ha de esperar en la pila el nombre del archivo, el buffer, las direcciones de los dos vectores y el número de elementos de los vectores (igual al número de líneas del archivo), nos encontramos con cinco valores en la pila, lo cual es demasiado y nos conduciría a una situación en que la mayor parte del código se ocupará de reordenar la pila. A esto se llama «ruido de pila». Aparte, tendríamos que gestionar los tres valores devueltos por `c-palabras`. Se impone por tanto otra solución. Una forma de mantener el problema en términos de sencillez es escribir una palabra que tome como argumento un número de columna y tras-

lade esa columna a un vector. Con dos pasadas al archivo colocamos cada columna en su lugar. Otra posibilidad es trasladar el archivo a una matriz. Extraer luego vectores columna es fácil. Además, no estamos limitados a dos columnas.

```
: f-a-matriz ( d d' d'' --)  \ archivo, buffer, matriz
    rot dup c-le@
    r/o open-file drop        \ d' d'' id
    over m-f 0 do             \ d' d'' id
        rot 2dup              \ d'' id d'  id d'
        dup c-l@ rot          \ d'' id d'  d' l id
        read-line             \ d'' id d' n flag 0
        drop drop over c-le!  \ d'' id d'
        dup c-bordes          \ d'' id d' d'
        dup c-#p 0 do
            c-ps              \ d'' id d' e
            dup dup c-lp      \ d'' id d' e e l
            c-a-entero        \ d'' id d' e v
            swap >r >r rot    \ id d' d''
            r> over ! cell+   \ id d' d''+c
            -rot r>           \ d'' id d' e
        loop
        drop -rot
    loop close-file 2drop ;
```

6. La palabra que escribiremos espera en la pila el nombre del archivo, el buffer donde serán puestas las líneas y la subcadena que va a filtrarse. Las líneas filtradas se imprimen simplemente. La salida podría redirigirse a otro archivo.

```
: f-grep ( d d' d'' --)
     cr
     rot dup c-le@ r/o
     open-file drop           \ d' d'' id
     begin
        rot 2dup              \ d'' id d' id d'
        dup c-l@ rot          \ d'' id d' d' l id
        read-line drop        \ d'' id d' n f
     while
        over c-le!            \ d'' id d'
```

```
        rot 2dup c-subc       \ id d' d'' f
        if
            over c-type cr    \ id d' d''
        then
        rot
     repeat drop
     swap close-file 2drop ;
```

7. Disponemos ya de una palabra que lee las columnas de un archivo de número enteros y los pasa a una matriz. Pero este ejercicio es diferente porque no se especifica que las columnas hayan de contener números. Nuestra implementación presupone que todas las líneas contendrán el mismo número de columnas. Espera en la pila los números de las columnas, el buffer donde serán leídas las líneas y el nombre del archivo. La idea es, si a es el número de la primera columna y b el de la segunda, leer cada línea en el buffer, avanzar hasta la palabra a e imprimirla y luego avanzar hasta la columna b e imprimirla. Para eso contamos con `c-ps`

```
: f-list2col ( a b d' d --)
    cr
    dup c-le@ r/o
    open-file drop            \ a b d' id
    begin
        2dup over c-l@ swap   \ a b d' id d' l id
        read-line drop        \ a b d' id n f
    while
        rot 2dup c-le! nip    \ a b id d'
        swap >r c-bordes      \ a b d'
        rot 2dup              \ b d' a d' a
        0 do c-ps loop        \ b d' a d''
        dup c-lp
        type 6 espacios       \ b d' a
        -rot swap 2dup        \ a d' b d' b
        0 do c-ps loop        \ a d' b d''
        dup c-lp type cr      \ a d' b
        swap r>               \ a b d' id
    repeat drop close-file
    3drop ;
```

Nos gustaría que las columnas quedaran alineadas. Para ello podríamos servirnos del vocabulario para cadenas que ya tenemos, o simplemente

imprimir tantos espacios en blanco a continuación de cada palabra como sean precisos para completar la anchura especificada. Escribimos otra versión para esta segunda solución:

```
: f-list2col-alineadas ( a b d' d --)
    cr
    dup c-le@ r/o
    open-file drop           \ a b d' id
    begin
        2dup over c-l@ swap   \ a b d' id d' l id
        read-line drop        \ a b d' id n f
    while
        rot 2dup c-le! nip    \ a b id d'
        swap >r c-bordes      \ a b d'
        rot 2dup              \ b d' a d' a
        0 do c-ps loop        \ b d' a d''
        dup c-lp tuck
        type 16 swap - spaces \ b d' a
        -rot swap 2dup        \ a d' b d' b
        0 do c-ps loop        \ a d' b d''
        dup c-lp type cr      \ a d' b
        swap r>               \ a b d' id
    repeat drop close-file
    3drop ;
```

Como se ve, la modificación es mínima.

8. Podemos usar el procedimiento del ejercicio anterior para tomar un archivo de texto que contiene columnas e imprimirlas alineadas. El problema es que o bien todas las columnas se imprimirán con la misma anchura o bien tendremos que especificar las anchuras en el código de la palabra que escribamos. Nada de esto es práctico y por eso lo que queremos es hacer una pasada previa al archivo y calcular la anchura máxima de cada columna, almacenando los datos en un vector, que usaremos después para imprimir todas las columnas alineadas y cada una con la anchura adecuada. La palabra espera en la pila el nombre del archivo, el buffer y un vector. En esta implementación se cuenta el número de palabras de cada línea, y por tanto no es preciso que todas las líneas tengan el mismo número de palabras, aunque sí lo es que el vector tenga una dimensión mayor o igual al número mayor de palabras que pueda encontrarse en una línea. Al principio, el vector de rellena con ceros.

```
: f-anchura-columnas ( d d’ d’’ N --)
    2dup 0v!! drop           \ d d’ d’’
    rot dup c-le@ r/o        \
    open-file drop           \ d’ d’’ id ----------------+
    begin                    \                           |
        rot 2dup             \ d’’ id d’ id d’           |
        dup c-l@ rot         \ d’’ id d’ d’ l id         |
        read-line drop       \ d’’ id d’ n f             |
    while                    \                           |
        over c-le!           \ d’’ id d’                 |
        rot 2dup swap        \ id d’ d’’ d’’ d’          |
        c-bordes dup c-#p    \ id d’ d’’ d’’ d’ n        |
        0 -rot               \ id d’ d’’ d’’ 0 d’ n      |
        0 do                 \ id d’ d’’ d’’ 0 d’ -----+ |
            c-ps dup c-lp    \ id d’ d’’ d’’ 0 d’ l    | |
            swap >r >r       \                         | |
            2dup cells +     \ id d’ d’’ d’’ 0 d’’+c   | |
            dup @ r> max     \ id d’ d’’ d’’ 0 d’’+c v | |
            swap ! 1+ r>     \ id d’ d’’ d’’ 0 d’ -----+ |
        loop                 \                           |
        3drop rot            \ d’ d’’ id ----------------+
    repeat 2drop close-file drop ;
```

En los comentarios al código, hemos conectado las líneas que contienen el estado de la pila al principio y al final de los dos bucles (el que recorre las líneas del archivo y el que recorre las palabras dentro de cada línea) para hacer explícito que el estado de la pila al principio de un bucle es (¡tiene que serlo siempre!) igual para cada iteración. Otra observación pertinente es que en la pila llegan a encontrarse hasta siete elementos, lo que es demasiado. Ahora bien, lo importante no es este número, sino el número de elementos sobre los que se opera en un momento dado, que es mucho menor.

9. Ahora ya podemos imprimir un archivo con sus columnas bien alineadas. En la primera pasada, rellenamos el vector X. Incrementamos en 4 el valor de cada elemento para dejar una separación entre columna y columna e imprimimos siguiendo el procedimiento que usamos en `f-list2col-alineadas`

```
: f-listcol ( d d’ d’’ N --)
    cr 2over 2over           \                                1
```

```
f-anchura-columnas  drop  \ d d' d''                          2
rot dup c-le@ r/o         \ d' d'' d l r/o                    3
open-file drop rot        \ d'' id d' -------------------------+  4
begin                     \                                 |  5
    2dup dup c-l@ rot     \ d'' id d' d' l id               |  6
    read-line drop        \ d'' id d' n f                   |  7
while                     \                                 |  8
    over c-le!            \ d'' id d'                       |  9
    rot swap              \ id d'' d'                       | 10
    2dup                  \ id d'' d'  d'' d'               | 11
    c-bordes dup c-#p     \ id d'' d'  d'' d' n             | 12
    0 do                  \ id d'' d'  d'' d' --------------+  | 13
        c-ps dup c-lp     \ id d'' d'  d'' d' l          |  | 14
        3dup              \ id d'' d'  d'' d' l  d'' d' l |  | 15
        tuck type         \                     d'' l    |  | 16
        swap @ 4 + swap - \                              |  | 17
        spaces drop       \ id d'' d'  d'' d'            |  | 18
        swap 1 cells +    \                              |  | 19
        swap              \ id d'' d'  d'' d' --------------+  | 20
    loop cr               \                                 | 21
    2drop rot swap        \ d'' id d'-------------------------+ 22
repeat                    \                                   23
2drop close-file 2drop ;  \                                   24
```

De nuevo, observamos que llegan a acumularse hasta 9 elementos en la pila. Sin embargo, nunca se opera con más de tres. En el bucle externo, es preciso preservar el identificador del archivo y las direcciones del buffer y el vector. En el bucle interno, es preciso operar con el buffer y el vector, pero también deben preservarse, incrementándose entre una iteración y la siguiente. De ahí el `2dup` de la línea 11 y el `3dup` de la 15. Hemos conectado las líneas 4-22 y las 13-20 para mostrar que el estado de la pila es el mismo en cada iteración. No obstante, esta es una palabra larga que podría factorizarse mejor. Por ejemplo, escribiendo una palabra independiente para imprimir una línea que tomase como argumentos la dirección de la cadena y la del vector. Eliminaríamos así el bucle interno.

2.9. Teclado y pantalla

1. `: at-fc swap at-xy ;`

```
: t-imprc ( ca f co --)
    at-fc emit ;
```

2.
```
: t-imprc* ( ca f co n --)
    -rot at-fc 0 do
        dup emit
    loop drop ;
```

3.
```
: t-ascii ( --)
    cr 6 spaces
    32 128 32  do
        dup 4 .r space
        dup emit
        dup 10 mod 0= if cr then
        1+
    loop cr ;
```

4.
```
: t-horizontal ( f c ancho --)
    >r 2dup at-fc [char] + emit r>
    swap 1+ swap
    2 - 0 do
        2dup at-fc [char] - emit
        1+
    loop at-fc [char] + emit ;

: t-vertical ( f c alto --)
    0 do
        2dup at-fc [char] | emit
        swap 1+ swap
    loop 2drop ;

: t-cuadrado ( f c ancho alto --)
    >r  3dup t-horizontal
    3dup rot  r> dup >r  + 1- -rot t-horizontal
    r> swap >r   3dup rot 1+ -rot 2 - t-vertical
    swap r> + 1- swap rot 1+ -rot 2 - t-vertical ;
```

5. Una vez dibujado el recuadro, el '*' se moverá entre las filas $f + 1$ y $f + alto - 2$ y las columnas $c + 1$ y $c + ancho - 2$. Encerraremos al asterisco en un recuadro cuya esquina superior izquierda tiene coordenadas $(0, 0)$ y 20 caracteres de ancho como de alto. Por tanto, el asterisco se moverá entre las filas 1 y 18, incluidas, y las columnas 1 y 18, incluidas.

```
: *on ( f c --) at-fc [char] * emit ;
: *off ( f c --) at-fc 32 emit ;

: t-prision ( --)
    page
    0 0 20 20 t-cuadrado
    10 10 2dup *on
    begin
        key
        dup 113 <>   \ 'q'
    while
        dup 97 = if \ 'a' izquierda
            drop dup 1 > if
                2dup *off 1- 2dup *on
            then
        else
        dup 115 = if \ 's' derecha
            drop dup 18 < if
                2dup *off 1+ 2dup *on
            then
        else
        dup 119 = if \ 'w' arriba
            drop over 1 > if
                2dup *off swap 1- swap 2dup *on
            then
        else
        dup 122 = if \ 'z' abajo
            drop over 18 < if
                2dup *off swap 1+ swap 2dup *on
            then
        else
            drop
        then then then then
    repeat 2drop page ;
```

6. Tanto el terminal ANSI como el terminal linux reconocen una serie de «secuencias de escape», que no son más que cadenas de caracteres el primero de los cuales es el carácter ESC (27). Algunas de estas secuencias permiten cambiar los atributos del texto que se presenta en pantalla. El formato de estas cadenas es `ESC[xm`, donde `x` es el carácter: `0` pa-

ra desactivar todos los atributos; 1 para texto resaltado; 4 para texto subrayado; 5 para parpadeo y 7 para video inverso. Es trivial escribir palabras para activar alguno de ellos, y para desactivarlos:

```
: ESC 27 emit ;
: t-desactiva ( --) ESC s" [0m" type ;
: t-resaltado ( --) ESC s" [1m" type ;
: t-subrayado ( --) ESC s" [4m" type ;
: t-parpadeo  ( --) ESC s" [5m" type ;
: t-inverso   ( --) ESC s" [7m" type ;
```

7. Guardaremos la secuencia de caracteres en una cadena. Comenzaremos rellenando el vector con la secuencia alfabética y luego produciremos unos cuantos intercambios aleatorios. Una vez hecho esto, dibujaremos el tablero y cada letra en su casilla, y entraremos en un bucle que gestione los movimientos.

```
: t-juego-horizontal
    s" +---+---+---+---+" type ;
: t-juego-vertical
    s" |   |   |   |   |" type ;
: t-juego-tablero
    page 0 0 at-fc
    4 0 do
        t-juego-horizontal cr
        t-juego-vertical   cr
    loop
    t-juego-horizontal cr
    s"   W"            type cr
    s" A   S  Ctrl-A"  type cr
    s"   Z"            type ;

: t-juego-casillas ( d --)    \ direccion de la cadena
    0 swap                    \ i d
    16 0 do
        over 4 /mod swap      \ i d f c
        4 * 2 + swap          \ paso a pantalla
        2 * 1 + swap          \ i d f c
        at-fc 2dup + c@ emit  \ i d
        swap 1+ swap          \ i+1 d
```

```
    loop 2drop ;

: rnd0-15 ( a --b b)          \ en el rango [0,15]
    899 * 32768 mod 16 mod dup ;
: t-juego-mezcla ( d s --)    \ cadena y semilla
    10 0 do
        rnd0-15 rnd0-15       \ d a b b
        >r rot tuck           \ a d b d
        dup >r
        + -rot + exch1
        r> r>                 \ d b
    loop 2drop ;

: t-juego-inicial ( d --n)    \ posicion inicial del hueco
    0                         \ d 0
    begin
        2dup + c@ [char] . <>  \ d 0 f
    while
        1+
    repeat nip ;

: t-juego-izquierda ( d n -- d n-1)
    dup 4 mod 0> if
        2dup 2dup              \ d n d n d n
        1- + -rot + exch1      \ d n
        1-                     \ d n-1
    then ;
: t-juego-derecha ( d n -- d n+1)
    dup 4 mod 3 < if
        2dup 2dup
        1+ + -rot + exch1
        1+
    then ;
: t-juego-arriba ( d n -- d n')
    dup 4 / 0> if
        2dup 2dup
        4 - + -rot +  exch1
        4 -
    then ;
: t-juego-abajo ( d n --d n')
    dup 4 / 3 < if
```

```
        2dup 2dup
        4 + + -rot + exch1
        4 +
    then ;

: t-juego-bucle ( d n --)      \ sale con Ctrl-A
    begin
        key dup 1 <>           \ d n k f
    while                      \ d n k
        dup 97 = if            \ d n k
            drop               \ d n
            t-juego-derecha    \ d n+1
        else
        dup 115 = if
            drop
            t-juego-izquierda
        else
        dup 119 = if
            drop
            t-juego-abajo
        else
        dup 122 = if
            drop
            t-juego-arriba
        else
            drop
        then then then then
        over t-juego-casillas
    repeat 3drop ;

: t-juego ( s -- )            \
    page t-juego-tablero
    s" diklbhjagmcenof."      \ s d n
    drop tuck
    swap t-juego-mezcla       \ d
    dup t-juego-casillas      \ d
    dup t-juego-inicial       \ d n
    t-juego-bucle
    page ;
```

8. Nos apoyaremos en los patrones de bits de la BIOS, donde están codifi-

cadas las figuras de cada carácter. Hemos escrito un pequeño programa en C bajo DOS que accede a la table ROM donde se encuentran estos patrones, y los hemos guardado en un archivo al que hemos llamado `patrones.dat`. Este archivo contiene un total de 4096 líneas, que son 256 bloques de 16 líneas. Cada línea contiene un número, de manera que un carácter, especificado mediante 16 bytes, puede verse como una matriz de 16 filas y 8 columnas. Los bits a 1 indican *pixels* iluminados. Para imprimir una versión *macro* de un carácter, en primer lugar cargamos `patrones.dat` en un vector (ya tenemos la palabra `f-a-vector`). Así, el carácter ASCII número n se encuentra encuentra codificado a partir de la posición $16n$.

```
: t-impr1 ( n --)              \ imprimer un byte
    7 0 do
        dup 2 mod
        swap 2 /
    loop
    8 0 do
        1 = if
            [char] # emit
        else
            32 emit
        then
    loop cr ;
: t-impr16 ( d --)             \ dada una direcci\'on
                               \ imprime el contenido y el de
                               \ los 15 bytes siguientes
    16 0 do
        dup @ t-impr1
        1 cells +
    loop drop ;

: t-cargar-patrones ( d d' d'' N --)
    f-a-vector ;

: t-grande  ( c d''  --)
    swap 16 * cells +
    t-impr16 ;
```

9. La palabra que vamos a escribir ha de esperar en la pila: un vector con los patrones de bits de los caracteres y la dirección de la cadena que

se va a imprimir. Por simplicidad, la cadena se mostrará en la primera línea del terminal, y la representación de cada carácter a partir de la fila 3 y la columna 0.

```
: t-deletrea0 ( c d'' --)
    3 0 at-fc t-grande ;
: t-deletrea ( d d'' N --)         \ espera cadena y patrones
    page drop
    over c-type 1 0 *on            \ d d''
    over c@ over                   \ d d'' c  d''
    t-deletrea0                    \ d d''
    swap 0                         \ d'' d contador
    begin
        key dup 1 <>               \ d'' d contador k f
    while                          \ d'' d contador k
        dup 97 = if                \ d'' d contador k
            drop                   \ d'' d contador
            dup 0 > if
                dup 1 swap *off
                1- dup 1 swap *on
                2dup + c@          \ d'' d contador c
                >r rot dup r> swap \ d contador d'' c d''
                t-deletrea0  -rot  \ d'' d contador
            then
        else
        dup 115 = if               \ d'' d contador k
            drop                   \ d'' d contador
            over c-le@             \ d'' d contador l
            over > if              \ d'' d contador
                dup 1 swap *off
                1+ dup 1 swap *on  \ d'' d contador
                2dup + c@          \ d'' d contador c
                >r rot dup r> swap \ d contador d'' c d''
                t-deletrea0 -rot
            then
        else
        then then
    repeat 2drop 2drop ;
```

2.10. Estructuras de datos

2.10.1. Pilas y colas

1. Al crearse, la primera celda indica el índice de la primera posición libre y la segunda el valor máximo del índice. Así es posible saber cuándo la pila está vacía o llena. En tiempo de ejecución, se deposita la dirección del primer elemento de la pila.

```
: pila create 0 , 3 - dup , cells allot
       does> 2 cells + ;
```

2. Será preciso escribir también palabras que indiquen si la pila se encuentra vacía o llena. Respecto a las palabras que apilan y desapilan, será preciso que dejen como resultado un *flag* que indique si la operación tuvo éxito o no. Finalmente, `p-tope` devuelve la dirección de la siguiente celda libre, `p-i+` incrementa el índice y `p-i-` lo decrementa. A efectos de depuración, conviene escribir otra palabra que imprima los elementos apilados, y otra que vacíe la pila.

```
: p-vacia? ( d --f)
    2 cells - @ 0= ;
: p-llena? ( d --f)
    dup 2 cells - @
    swap 1 cells - @ > ;
: p-# ( d --n)  \ numero de elementos apilados
    2 cells - @ ;
: p-tope ( d --d')
    dup 2 cells - @ cells + ;
: p-i+ ( d --)
    2 cells - dup @ 1+ swap ! ;
: p-i- ( d --)
    2 cells - dup @ 1- swap ! ;

: >p ( n d --)  \ sin seguridad
    tuck p-tope ! p-i+ ;
: p>  ( d -- n) \ sin seguridad
    dup p-tope 1 cells - @
    swap p-i- ;

: p. ( d --)
```

```
    dup p-vacia? if drop exit then
    cr dup 2 cells - @
    0 do
        @+c . cr
    loop ;
: p-0 ( d --)
    2 cells - 0 swap ! ;
```

3. ```
   : p-drop ( d --)
       p> drop ;
   : p-dup  ( d --)
       dup p-tope 1 cells - @
       swap >p ;
   : p-swap ( d --)
       dup dup p> swap p>
       >r over >p r> swap >p ;
   ```

4. Implementaremos la cola FIFO mediante 13 palabras. En primer lugar, crear la cola:

   ```
   : fifo create dup , 0 , 0 , 0 , 4 - allot
          does> 4 cells + ;
   ```

   Al crearse, las cuatro primeras celdas están reservadas: una para el total de elementos, otra para el número actual de elementos, otra para el puntero de lectura y la cuarta para el puntero de escritura.

5. Las doce palabras para gestionar la cola son: `fifo-L?` devuelve el puntero de lectura; `fifo-E?` devuelve el puntero de escritura; `fifo-N?` devuelve el número máximo de elementos que la cola puede almacenar. `fifo-#?` devuelve el número de elementos contenidos efectivamente en la cola; `fifo-L` efectúa una lectura; `fifo-E` efectúa una escritura; `fifo-#+` incrementa el número de elementos; `fifo-#-` decrementa el número de elementos; `fifo-L+` incrementa el puntero de lectura; ; `fifo-E+` incrementa el puntero de escritura. Las operaciones de lectura dejan en la pila: un *flag* a cero si la cola estaba vacía; el valor leído y un *flag* a -1 en caso contrario. Las operaciones de escritura consumen su argumento en cualquier caso, y dejan un *flag* a 0 o -1 según la operación fracasó o tuvo éxito. Serán útiles otras dos palabras adicionales que nos digan si la cola se encuentra llena o vacía, y una tercera a efectos de depuración que imprima los elementos encolados.
   ```

```
: fifo-L? ( d -- n)
   2 cells - @ ;
: fifo-E? ( d --n)
   1 cells - @ ;
: fifo-N? ( d --N)
   4 cells - @ 4 - ;
: fifo-#? ( d --n)
   3 cells - @ ;
: fifo-L+ ( d --)
   2 cells - dup dup @ 1+ swap
   fifo-N? mod swap ! ;
: fifo-E+ ( d --)
   1 cells - dup dup @ 1+ swap
   fifo-N? mod swap ! ;
: fifo-#+ ( d --)
   3 cells - dup @ 1+ swap ! ;
: fifo-#- ( d --)
   3 cells - dup @ 1- swap ! ;
: fifo-llena? ( d -- f)
   dup fifo-N? swap fifo-#? = ;
: fifo-vacia? ( d -- f)
   fifo-#? 0= ;
: fifo-L ( d --n)
   dup fifo-vacia? if
       drop 0
   else
       dup dup fifo-L? cells +
       @ swap -1 swap
       dup fifo-L+ fifo-#-
   then ;
: fifo-E ( n d --)
   dup fifo-llena? if
       2drop 0
   else
       dup dup fifo-E? cells +
       rot swap !
       dup fifo-E+ fifo-#+ -1
   then ;
: fifo. ( d --)
   cr dup dup fifo-L? swap fifo-E?
   begin
```

```
    2dup <
while
    -rot 2dup cells + @ . cr
    1+ rot
repeat drop ;
```

2.10.2. Memoria dinámica

El objeto de esta sección es escribir un gestor de memoria dinámica que será usado después al tratar de otras estructuras de datos, como listas y árboles. El fragmento de memoria que se va a gestionar se toma del propio diccionario. En cuanto a la gestión en sí, se hará mediante un par de listas enlazadas: una para los fragmentos libres y otra para los fragmentos ocupados. Cuando se hace una solicitud de memoria, se recorre la lista de fragmentos libres hasta encontrar uno mayor que la cantidad solicitada. Entonces ese fragmento se divide en dos: una parte se reserva y se ñade a la lista de ocupados mientras que la otra permanece en la lista de libres, actualizando su tamaño. Liberar memoria es sencillo: el fragmento se traspasa de una lista a otra.

Cada fragmento contiene no sólo la cantidad de memoria libre u ocupada, sino un puntero al fragmento siguiente de su misma clase y el tamaño mismo.

Finalmente, de toda la cantidad que va a gestionarse será preciso reservar cuatro celdas: dos punteros para el principio y final de la lista de huecos libres y otros dos para la lista de fragmentos ocupados.

La preparación del espacio en el diccionario se hace mediante las palabras siguientes:

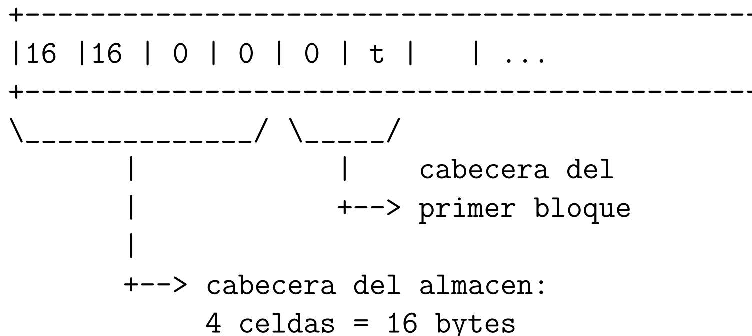
```
: almacen create
          here 4 cells + dup , ,
          0 , 0 ,
          0 , 6 cells - ,
          allot
          does> ;
```

Como se ve, las direcciones que se guardan en las cabeceras son direcciones absolutas. La creación de un almacén comienza con la creación de la entrada en el diccionario y la compilación de los punteros, donde el 0 equivale a *nil*. Entonces, el espacio queda dividido en una primera parte con seis celdas y una segunda con el resto hasta completar el tamaño que se especificó. De la primera parte, cuatro celdas indican el primer bloque libre, el último bloque libre, el primer bloque ocupado y el último bloque ocupado. Dos más son la

cabecera del primer bloque libre, que al iniciar el almacén es el único existente y tiene por tamaño el que se especificó menos las seis celdas nombradas. Para evitar colisión de nombres, usaremos en todas las palabras el prefijo `md-`. La dirección de un almacén se indica en los comentarios de pila mediante la letra 'a'.

```
: md-PL ( a --d)
    @ ;
: md-UL ( a --d)
    1 cells + @ ;
: md-PO ( a --d)
    2 cells + @ ;
: md-UO ( a --d)
    3 cells + @ ;
```

`PL` indica la dirección del bloque **P**rimero **L**ibre. `UL` la dirección del último bloque libre. `PO` la dirección del primer bloque ocupado y `UO` la dirección del último bloque ocupado.



Respecto a un bloque determinado, interesa la dirección del primer byte usable. Dada esta dirección, las dos palabras siguientes obtienen el tamaño del bloque y el desplazamiento del bloque siguiente de su lista:

```
: md-S ( d -- d')
    2 cells - @ ;            \ siguiente
: md-T ( d --t)
    1 cells - @ ;            \ taman~o
```

Lo siguiente que hemos de procurar es, dada una petición, recorrer la lista de bloques libres hasta encontrar uno lo bastante grande para dividirlo en dos: una parte que se añadirá a la lista de ocupados y otra parte que queda en la de libres, pero con el tamaño modificado. Como la parte que se ocupa

ha de contener también la cabecera, que son dos celdas, el tamaño libre ha de exceder en dos celdas al tamaño que se solicita.

Así pues, escribiremos dos palabras: una que busca un hueco libre lo bastante grande y otra que hace la división y actualiza las listas. `md-hueco` espera en la pila un tamaño y la dirección de un almacén y busca un bloque de ese almacén donde acomodarlo. Si lo encuentra, devuelve la dirección del bloque. Si no, devuelve un 0. `md-cabe?` espera un tamaño y la dirección de un bloque, e indica si ese tamaño puede acomodarse en el bloque.

```
: md-cabe? ( t d --f)
    md-T 2 cells - < ;
: md-hueco ( t d -- d' )
    2dup md-cabe? if
        nip
    else
        dup 0= if
            nip
        else
            md-S recurse
        then
    then ;
```

Una vez encontrado un bloque capaz de acomodar el tamaño solicitado, es preciso:

1. Dividirlo en dos partes, creando la cabecera del nuevo bloque ocupado
2. Añadir este nuevo bloque ocupado a la lista de bloques ocupados
3. Actualizar el puntero a «último bloque ocupado»
4. Actualizar el tamaño del bloque libre original con el valor que quede después de sustraerle el tamaño pedido

La palabra `md-reserva` espera en la pila la dirección del almacén, la dirección del bloque libre que puede acomodar la reserva solicitada y el tamaño que se solicita reservar.

```
: md-reserva  ( a d r --)       \                1
    dup >r                      \ para despues   2
    2dup over md-T swap - +     \ a d r d'       3
    tuck 1 cells + !            \ a d d'         4
```

```
0 over !                     \ a d d’         5
rot dup md-UO >r             \                6
-rot dup r> !                \ a d d’         7
rot 3 cells + over swap !  \ d d’           8
swap r>                      \ d’ d r         9
over md-T swap - 2 cells - \               10
swap 1 cells + ! ;           \ d’            11
```

En la línea 3 se calcula la dirección del nuevo bloque ocupado. En la 4 se rellena con el tamaño la segunda celda de la cabecera. En la línea 5 se rellena la primera celda de la cabecera, que apunta al bloque siguiente ocupado. Como el bloque que se está creando es el último, el puntero se pone a 0. En las líneas 6-7 se obtiene de la cabecera del almacén la dirección del último bloque ocupado y se hace apuntar al recién creado, que pasa a ser el último. En la línea 8 se actualiza el puntero al último bloque ocupado en la cabecera del almacén. Finalmente, se actualiza el tamaño libre en el bloque del que se ha tomado la reserva.

La siguiente operación básica consiste en la liberación de un bloque. Para ello basta traspasarlo de la lista de ocupados a la lista de libres, añadiéndolo al final de ésta. A su vez, este traspaso consta de varias operaciones elementales:

1. Encontrar el bloque anterior al que es preciso liberar.
2. Enlazar éste con el bloque siguiente al que deseamos liberar.
3. Traspasar el bloque ocupado a la lista de bloques libres.

`md-primero-ocupado?` comprueba si el bloque que se quiere liberar es el primero de la lista, devolviendo un *flag*. `md-anterior-ocupado?` espera dos direcciones: una de la lista y otra del bloque que se quiere liberar, y devuelve la dirección del bloque anterior al que se quiere liberar. Funciona recursivamente. `md-busca` espera la dirección del almacén y la dirección del bloque que se quiere liberar. Si este bloque es el primero, se devuelve la dirección del almacén. Si no se toman como argumentos la dirección del primer bloque ocupado y la dirección del que se quiere ocupar y se busca a partir de ahí la del bloque anterior mediante la función recursiva previa.

```
: md-primero-ocupado? ( a d --f)
    swap md-PO = ;
: md-anterior-ocupado? ( d d’ -- d’’)
    2dup swap md-S =
    if
```

```
        drop
    else
        swap md-S
        swap recurse
    then ;
: md-busca ( a d -- d')
    2dup md-primero-ocupado?
    if
        drop
    else
        swap md-PO
        swap md-anterior-ocupado?
    then ;
```

Enlazar el bloque anterior al que se quiere liberar con el siguiente es fácil. `md-enlaza` espera la dirección del bloque anterior y la dirección del que se quiere liberar.

```
: md-enlaza ( d d' --)
    md-S swap ! ;
```

Finalmente, hay que integrar el bloque liberado en la lista de libres. `md-traspasa` toma la dirección de un bloque ocupado y lo enlaza al último de la lista de libres. Espera también la dirección del almacén. Es preciso: poner a 0 el enlace del bloque que se traspasa, enlazar el último de la lista de libres con el que se traspasa y modificar en la cabecera del almacén el campo correspondiente al último elemento de la lista de libres.

```
: md-traspasa ( a d --)
    dup 0 swap !
    over md-UL over
    swap ! swap ! ;
```

Faltan dos cosas: construir una interfaz `malloc` y `free` usando estas palabras, lo cual se reduce a un poco de «código argamasa» y escribir un desfragmentador, ya que si fallase `malloc` pudiera ser que desfragmentando la memoria pudiese satisfacerse la petición. En efecto, si se agota el almacén mediante una serie de peticiones de pequeño tamaño y luego se libera cada fragmento estaremos en un punto en que el almacén está libre completamente pero no hay bloque que pueda usarse. Por eso es preciso escribir un desfragmentador que fusione los bloques contiguos libres en uno solo.

No seguiremos adelante por varias razones: 1) no complicar lo que es un simple ejercicio; 2) porque los sistemas ya cuentan con un gestor de memoria dinámica; 3) porque nos llevaría más lejos de lo que deseamos definir variables tipo puntero y definir las operaciones con punteros y 4) porque acabamos de ver cómo gestionar una lista.

En efecto: es posible usar punteros y un gestor de memoria genérico para crear y manipular listas o bien es posible escribir directamente sobre el espacio dado un gestor de listas. Eso es lo que hemos hecho para construir el gestor de memoria. Y es evidente que igual que hemos hecho para un fin puede hacerse para otro.