Primer Concurso Local de Programación ACM ICPC
Universidad Autónoma Metropolitana Unidad Azcapotzalco

Examen final, 21 de Octubre de 2004

Instrucciones: Abajo encontrará los enunciados de tres problemas. Para cada problema que resuelva deberá dejar en el directorio de trabajo `c:\acmNN` los archivos de un programa (tanto en código fuente como en ejecutable). El nombre de estos archivos es de la forma `xxxNN.zzz`, donde `xxx` es el nombre del problema, `NN` es un número de identificación que se le dará al comenzar el examen y `zzz` es la extensión, según el tipo de archivo. Si así lo desea, puede resolver cada problema en un lenguaje distinto. Su programa deberá leer e imprimir exactamente los datos que se indican (ni más ni menos). Recuerde que en esta ocasión, sus programas deberán trabajar con **archivos de texto** (cuyos nombres aparecerán en la descripción de cada uno de los problemas). En particular, sus programas no deberán borrar la pantalla, escribir ningún tipo de letrero, usar la unidad `screen`, la biblioteca `conio`, etc. Cada problema tiene un valor de 100 puntos, los cuales se obtendrán de las salidas que su programa entregue para cada una de 10 entradas distintas. Que su programa funcione con el ejemplo no quiere decir que su programa funcionará siempre.

Cuestiones técnicas: En todas las computadoras debe existir un directorio de trabajo llamado `c:\acmNN` (en caso contrario, éste se puede crear con las instrucciones (a) `c:` (b) `cd c:\` (c) `mkdir acmNN` (d) `cd acmNN`). Quienes programen en C o C++ deberán usar el **DJGPP**, el cual se puede ejecutar con la instrucción `rhide` desde el directorio de trabajo. Quienes programen en Pascal deberán usar el **Borland Turbo Pascal**, el cual se puede ejecutar con la instrucción `turbo` desde el directorio de trabajo. Quienes programen en Java deberán usar el **Sun Java SDK**. Si algo no funciona, repórtelo de inmediato.

```
/* Los organizadores le deseamos éxito en su examen */
```

Problema 1: Dando cambio en el autobús

Código fuente: `dcaNN.c, dcaNN.cpp, dcaNN.java, dcaNN.pas`
Archivos de entrada y de salida: `dca.ent` y `dca.sal`
Programa ejecutable: `dcaNN.exe, dcaNN.class`

Si los conductores de autobús hicieran bien su trabajo (y no nos referimos al buen conducir, para lo cual hay poca esperanza) debieran de dar el cambio usando tan pocas monedas como sea posible. Así no molestarían a los clientes ni se les acabarían las monedas. En nuestro

país tenemos monedas de 1, 2, 5, 10 y 20 pesos, así que si quisieramos dar 47 pesos de cambio, lo mejor que podemos hacer es dar dos monedas de 20, una de 5 y una de 2. Sin embargo, en algún otro país podrían tener otros valores para las monedas, como por ejemplo de 1, 3 y 5. Escriba un programa que, dados la cantidad **c** de cambio a dar, la cantidad **m** de monedas distintas de las que se dispone, y los valores $\mathbf{v}_1,...,\mathbf{v}_m$ de esas monedas (ordenados ascendentemente) encuentre una forma de dar el cambio usando tan pocas monedas como le sea posible. Suponga que siempre habrá monedas de valor 1, así que siempre se puede dar cualquier cantidad de cambio.

Entrada: El archivo `dca.ent` contendrá dos enteros **c** y **m** (separados por espacios) seguidos de una lista de **m** enteros $\mathbf{v}_1$, ..., $\mathbf{v}_m$ (separados por espacios) tales que $1 \leq m \leq 100$ y $1 = \mathbf{v}_1 < \mathbf{v}_2 < ... < \mathbf{v}_m \leq \mathbf{c} \leq 1000$.
Salida: El archivo `ccc.sal` deberá contener una lista de **m** enteros $\mathbf{n}_1$, $\mathbf{n}_2$, ..., $\mathbf{n}_m$ (todos ellos no negativos y separados por espacios) tales que $\mathbf{n}_1 \mathbf{v}_1 + \mathbf{n}_2 \mathbf{v}_2 + ... + \mathbf{n}_m \mathbf{v}_m = \mathbf{c}$ (es decir, para $1 \leq \mathbf{i} \leq \mathbf{m}$, $\mathbf{n}_i$ es la cantidad de monedas de valor $\mathbf{v}_i$ para dar el cambio **c**) y que la cantidad total de monedas $\mathbf{n}_1 + \mathbf{n}_2 + ... + \mathbf{n}_m$ sea tan pequeña como le sea posible.
Evaluación: Si **n** es el número mínimo de monedas necesarias, entonces se otorgarán $10\mathbf{n} / (\mathbf{n}_1 + \mathbf{n}_2 + ... + \mathbf{n}_m)$ puntos (sólo si la respuesta es correcta). En los dos ejemplos de abajo n = 3, así que el primer archivo de salida obtendría 10 puntos, el segundo obtendría 6 puntos y el tercero obtendría sólamente un 1 punto.

Ejemplos de archivo de entrada	Ejemplos de archivo de salida
`9 3 1 3 5`	`0 3 0`
`9 3 1 3 5`	`4 0 1`
`9 3 1 3 5`	`9 0 0`

Problema 2: Collar con cuentas de colores

Código fuente: `cccNN.c, cccNN.cpp, cccNN.java, cccNN.pas`
Archivos de entrada y de salida: `ccc.ent` y `ccc.sal`
Programa ejecutable: `cccNN.exe, cccNN.class`

Cierta persona tiene un collar **circular** con cuentas de colores: hay cuentas **a**marillas, **b**lancas, **c**afés, etc., de modo que podemos describir el collar comenzando en cualquier lugar del mismo, avanzando siempre en la misma dirección e indicando la inicial del color correspondiente (una letra minúscula de la `a` a la `z`, sin incluir a las letras con acento ni a la ñ) hasta llegar a la última cuenta. Esta persona quiere responder **tres** preguntas que le hicieron acerca de su collar. La primera pregunta es ¿cuántos colores distintos hay en su collar? (llamemos **c** a

esta cantidad). La segunda pregunta es ¿cuál es la longitud del segmento más largo de su collar tal que todas las cuentas son del mismo color? (llamemos **m** a esta cantidad). Ahora imagine que puede cortar el collar en cualquier posición para luego sacar todas las cuentas del mismo color que estén inmediatamente a la izquierda del punto de corte y todas las cuentas del mismo color que estén inmediatamente a la derecha del punto de corte (por supuesto, estos dos colores pueden ser iguales o diferentes). Así, la tercera pregunta es: De entre todos los posibles lugares de corte ¿cuál es la mayor cantidad de cuentas que se pueden sacar del collar? (llamemos **t** a esta cantidad). Escriba un programa que ayude al dueño del collar a calcular las tres cantidades **c**, **m** y **t**. En el collar descrito en el ejemplo de abajo hay 3 colores distintos, hay un segmento de 4 cuentas del mismo color, y se puede cortar entre la sexta y la séptima posición para sacar 5 cuentas (también se puede cortar en otros dos lugares para sacar el mismo número de cuentas).

Entrada: El archivo `ccc.ent` contendrá una cadena de caracteres (todos ellos letras minúsculas) cuya longitud está entre 2 y 1000.
Salida: El archivo `ccc.sal` deberá contener los tres números enteros **c**, **m**, **t**, separados por espacios.
Evaluación: 2 puntos por **c**, 3 puntos por **m** y 5 puntos por **t**.

Ejemplo de archivo de entrada	Ejemplo de archivo de salida
`uamaaauumaummuauamumummmmua`	`3 4 5`

Problema 3: Dos robots descompuestos

Código fuente: `drdNN.c, drdNN.cpp, drdNN.java, drdNN.pas`
Archivos de entrada y de salida: `drd.ent` y `drd.sal`
Programa ejecutable: `drdNN.exe, drdNN.class`

Un profesor del Departamento de Robótica de cierta universidad tiene dos robots para realizar ciertos experimentos de inteligencia artificial. La idea es programar a esos robots para que puedan salir de un laberinto. De manera más bien desafortunada, sus dos robots están descompuestos y ya no es posible reprogramarlos, además de que debido al bajo presupuesto con el que cuenta, tampoco puede comprar unos nuevos. El primer robot, **Diestro**, sólo puede ejecutar el siguiente programa: Comienza avanzando hacia la **derecha**. Mientras no llegue a la salida, avanzará siempre en la misma dirección, a menos que encuentre una pared en su camino, momento en el cual girará 90 grados a la **derecha** (sentido horario) y continuará avanzando. De manera similar, el segundo robot, **Siniestro**, sólo puede ejecutar el siguiente programa: Comienza avanzando hacia **abajo**. Mientras no

llegue a la salida, avanzará siempre en la misma dirección, a menos que encuentre una pared en su camino, momento en el cual girará 90 grados a la **izquierda** (sentido antihorario) y continuará avanzando. Escriba un programa que, dada la descripción de un laberinto, diga cuántos pasos se tarda cada robot en llegar de la entrada (que siempre estará en la esquina superior izquierda) a la salida (que siempre estará en la esquina inferior derecha), o que indique que no pueden llegar (por ejemplo, si se queda dando vueltas en algún lado). **Se considera un paso el avanzar una unidad**. Sea **d** el número de pasos que necesita Diestro y **s** el que necesita Siniestro. El laberinto es una cuadrícula que mide **m** unidades horizontales y **n** unidades verticales. Cada cuadro de la cuadrícula puede tener pared en cualquiera de las cuatro direcciones arriba, abajo, derecha e izquierda, y esto se indica con la suma de los números 1, 2, 4, 8. Por ejemplo, si un cuadro tiene el número 5, esto significa que tiene pared arriba y a la derecha, y que no tiene pared abajo ni a la izquierda. El laberinto no tiene porqué estar rodeado por una pared, y **si algún robot se sale del laberinto se debe considerar que no puede llegar a la salida** (además de que el pobre profesor tendrá que correr a perseguirlo).

Entrada: El archivo `drd.ent` contendrá dos números enteros **m** y **n** (separados por un espacio y con valores entre 2 y 100), seguidos de **m** renglones (de arriba hacia abajo), cada uno con **n** descripciones de cuadros (separadas por espacios y de izquierda a derecha).
Salida: El archivo `drd.sal` deberá contener los dos números enteros **d** y **s** separados por espacios. Si algún robot no puede llegar, se indicará con un `-1`.
Evaluación: 5 puntos por el valor de **d** y 5 puntos por el valor de **s**.

Ejemplo de archivo de entrada	Ejemplo de archivo de salida
4 5 8 2 4 9 5 0 1 4 8 4 4 8 6 0 0 2 0 1 0 2	-1 7

Para su comodidad, este es el diagrama del laberinto descrito arriba: