Chocolate - Solución

Este problema se parte en dos partes: recoger todos los zumbadores y repartirlos en las filas inferiores hasta que se acaben.

Para recogerlos solo hay que hacer una ciclo mientras estemos junto a un zumbador, cuidando las validaciones de no chochar con la pared.

Para dejarlos debemos hacer otro ciclo mientras tengamos zumbadores en nuestra mochila

```
iniciar-programa
    inicia-ejecucion
        mientras junto-a-zumbador hacer inicio
            coge-zumbador;
            si frente-libre entonces avanza;
        fin;

        gira-izquierda;
        mientras frente-libre hacer avanza;
        gira-izquierda;
        mientras frente-libre hacer avanza;
        gira-izquierda;

        mientras algun-zumbador-en-la-mochila hacer inicio
            deja-zumbador;
            si frente-libre entonces inicio
                avanza;
            fin sino inicio
                gira-izquierda;
                si frente-libre entonces inicio
                    avanza;
                    gira-izquierda;
                    mientras frente-libre hacer avanza;
                    gira-izquierda;
                    gira-izquierda;
                fin;

            fin;
        fin;
        apagate;
    termina-ejecucion
finalizar-programa
```

Arcoíris - Solución

Observando como luce el arcoíris una vez que se dibuja, podemos darnos cuenta de que en la columna central del mundo aparecen los números del 1 al 7 de manera consecutiva, de tal manera que el $7 $ aparece en la celda más al norte de dicha columna. Vemos además que desde la columna central salen escaleras descendientes a la izquierda y a la derecha: del 1 de la columna central sale una escalera descendiente a la derecha y otra a la izquierda, del 2 de la columna central sale una escalera descendiente a la derecha y otra a la izquierda, y así consecutivamente.

El problema se reduce entonces a encontrar la columna central (para esto basta ver que la columna central es igual a la altura del mundo) y posteriormente bajar a la posición donde debe ir el 1 de esta columna, dibujar sus dos escaleras para después subir a la posición del 2 de la columna central y dibujar sus escaleras y así consecutivamente.

A continuación se muestra una forma de encontrar la columna central.

```
define-nueva-instruccion centro como inicio
     si frente-libre entonces inicio
         avanza;
         centro;
         avanza;
     fin sino inicio
         este;
     fin;
 fin;
```

Y esta es la implementación para dibujar una de las escaleras del arcoíris. Nota que usamos recursividad para volver a la columna central.

```
define-nueva-instruccion escalera-izq(n) como inicio
     si izquierda-libre entonces inicio
         avanza;
         sur;
         avanza;
         oeste;
         deja(n);
         escalera-izq(n);
         norte;
         avanza;
         este;
         avanza;
     fin;
 fin;
```

Líneas - Solución

Calculemos la paridad de un montón de beepers sin hacer movimientos.

El código es este:

```
void checkParity() {
        if (nextToABeeper) {
            pickbeeper();
            if (nextToABeeper) {
                pickbeeper();
                checkParity();
                putbeeper();
            } else {
                east();
            }
            putbeeper();
        } else {
            west();
        }
    }
```

Observemos que para dibujar N puntos en a lo más $2N$ movimientos, solo podemos recorrer el mundo a lo mas dos veces. Consideremos solo los montones de tamaño impar en la primer corrida del mundo (de izquierda a derecha).

Consideremos el primer montón de beepers (solo consideramos los de tamaño impar por el momento). Al dibujar esta línea, podemos encontrarnos con otro montón de beepers (solo consideramos tamaños impares). Su línea correspondiente comparte cierta cantidad de posiciones con la que estamos pintando actualmente. Pintemos la que termina mas a la derecha (esto es fácil si vemos cuantos beepers del nuevo montón hacen falta considerar).

Para dibujar las líneas, basta con usar una función recursiva con un parámetro que nos sirva como contador.

El código es este:

```
void extendOddLine(x) {
        if(nextToABeeper) {
            pickbeeper();
            extendOddLine(succ(x));
        } else {
            oddLine(x);
        }
    }

    void oddLine(x) {
        if (nextToABeeper) {
```

```
            checkParity();
            if (facingEast) {
                // Impar
                iterate(x) if(nextToABeeper) pickbeeper();
                extendOddLine(x);
            } else {
                east();
                if(frontIsClear) {
                    move();
                    if(!iszero(x)) oddLine(pred(x));
                    else oddLine(x);
                }
            }
        } else {
            if (frontIsClear) {
                if(!iszero(x)) {
                    putbeeper();
                    move();
                    oddLine(pred(x));
                } else {
                    move();
                    oddLine(x);
                }
            } else if(!iszero(x)) putbeeper();
        }
    }
```

Al terminar de pintar las líneas de tamaño impar, estamos en el extremo derecho del mundo y ahora el problema es pintar las líneas de tamaño par en una corrida de derecha z izquierda. Observemos que es casi el mismo problema que acabamos de resolver, solo basta con considerar únicamente montones de tamaño par.

El código es este:

```
void oddLine(x) {
        if (nextToABeeper) {
            checkParity();
            if (facingEast) {
                // Impar
                iterate(x) if(nextToABeeper) pickbeeper();
                extendOddLine(x);
            } else {
                east();
                if(frontIsClear) {
                    move();
                    if(!iszero(x)) oddLine(pred(x));
                    else oddLine(x);
```

```
            }
        }
    } else {
        if (frontIsClear) {
            if(!iszero(x)) {
                putbeeper();
                move();
                oddLine(pred(x));
            } else {
                move();
                oddLine(x);
            }
        } else if(!iszero(x)) putbeeper();
    }
}

void extendEvenLine(x) {
    if(nextToABeeper) {
        pickbeeper();
        extendEvenLine(succ(x));
    } else {
        evenLine(x);
    }
}

void evenLine(x) {
    if (nextToABeeper) {
        checkParity();
        if (facingWest) {
            // Par
            iterate(x) if(nextToABeeper) pickbeeper();
            extendEvenLine(x);
        } else {
            west();
            if(frontIsClear) {
                move();
                if(!iszero(x)) evenLine(pred(x));
                else evenLine(x);
            }
        }
    } else {
        if (frontIsClear) {
            if(!iszero(x)) {
                putbeeper();
                move();
                evenLine(pred(x));
```

```
            } else {
                move();
                evenLine(x);
            }
        } else if(!iszero(x)) putbeeper();
    }
}
```

Casilleros - Solución

La observación crucial a este problema es darse cuenta que las casillas que al final estarán prendidas son las que corresponden a los cuadrados perfectos (enumerando desde cero la primera casilla). Para notar lo anterior basta con tener en cuenta que cada interruptor va a ser cambiado de estado únicamente por sus divisores. Después de esto, sabemos que permanecerá prendido si y solo si es cambiado un número impar de veces. Por lo tanto, los interruptores que tienen un número impar de divisores serán los que terminarán encendidos. Estos son los números cuadrados.

Ahora bien, la idea de solución consiste en pasar de un número cuadrado perfecto al siguiente solamente avanzando hacia la derecha. Encontraremos ahora una forma de llegar al siguiente cuadrado sin regresar.

Imaginemos que estamos en el n-ésimo cuadrado y queremos llegar al $(n+1)$-ésimo. Esto quiere decir que nos encontramos en el interruptor número n^2 y queremos llegar al interruptor $(n+1)^2$. Para hacer eso necesitamos avanzar exactamente $(n+1)^2 - n^2 = 2n+1$ pasos. Notemos que $2n+1$ es la secuencia de números impares. Es decir, para llegar a la casilla uno debemos avanzar 1, para llegar de la casilla uno a la cuatro nos movemos 3, de la cuatro a la nueve nos movemos 5 y así sucesivamente.

Con todo lo anterior, podemos programar una función que nos diga en un parámetro la cantidad de pasos que debemos movernos para llegar al siguiente interruptor que debemos prender y volver a llamar esta función con el nuevo número de pasos que necesitamos (el siguiente número impar).

El código se ve así:

```
define-nueva-instruccion prendeInterruptores(pasos) como inicio
   avanzaNpasosPonZumbador(pasos);
   prendeInterruptores(sucede(sucede(pasos)));
fin;
```

Soldados - Solución

Las casillas del mundo que inicialmente tengan zumbadores en ellas deberán quedar con la misma cantidad de zumbadores al final del programa, por lo que solo deberemos por preocuparnos por aquellas casillas con 0 zumbadores. Entiéndase por casillas adyacentes aquellas que se encuentran al oeste y al sur de cada casilla.

Para que una casilla (x, y) sea mayor que las casillas adyacentes $(x-1, y)$ y $(x, y-1)$, es suficiente con obtener el valor mayor de ambas casillas y sumarle 1. Formalmente: $f(x,y) = \max(f(x-1,y), f(x,y-1)) + 1$, donde $f(i, j)$ representa el valor a asignar en la casilla (i, j).

Figure 1: 1

Se suma 1 al valor máximo de ambas casillas porque es el número mínimo necesario para que la casilla (x, y) sea mayor que sus 2 casillas adyacentes. Si decidiéramos sumarle un valor $n > 1$ al máximo de ambas casillas, pudiéramos encontrarnos con el siguiente mundo:

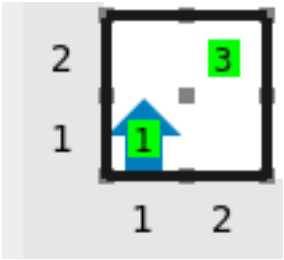

Figure 2: 2

En donde la única solución es sumarle 1 al máximo de las casillas adyacentes.

¿Qué sucede cuando hay una pared en la casilla $(x-1, y)$ o $(x, y-1)$?

Podemos asumir que si no es posible acceder a las casillas adyacentes, el valor de dichas casillas es 0, entonces solo nos importará sumarle 1 al valor de las casillas a las que sí se puede acceder.

El siguiente código asigna el valor a una casilla.

```
void rellenaCasilla() {
```

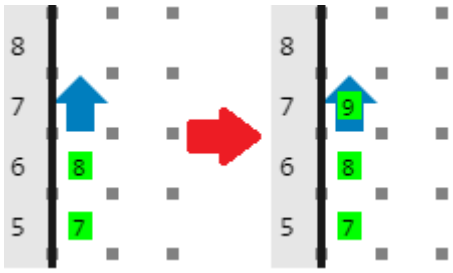

Figure 3: 3

```
        if (notNextToABeeper) {
            turnleft();
            checa();
            turnleft();
            checa();
            backturn();
            putbeeper();
        }
    }
```

La instrucción `checa` se encarga de verificar si existe una pared en las casillas adyacentes, de no ser así, entonces copia el valor de dichas casillas, y deja en la casilla (x, y) el valor mayor.

```
void checa() {
        if(frontIsClear) {
            move();
            backturn();
            pasa(0);
            backturn();
        }
    }
```

La instrucción `pasa` se encarga de crear una copia del montón de beepers de una casilla adyacente. Finalmente, la instrucción `rellenaCasilla` deja un zumbador extra para sumar 1 al valor mayor obtenido.

```
void pasa(x) {
        if (nextToABeeper) {
            pickbeeper();
            pasa(succ(x));
        } else {
            iterate(x)
                putbeeper();
            move();
            iterate(x)
```

```
            if(nextToABeeper)
                pickbeeper();
        iterate(x)
            putbeeper();
    }
}
```

Para calcular el valor en la casilla (x, y) es necesario primero calcular el valor de las casillas $(x-1, y)$ y $(x, y-1)$, y a su vez, para calcular el valor de la casilla $(x-1, y)$ es necesario calcular primero el valor de las casillas $(x-2, y)$ y $(x-1, y-1)$. Para asegurarnos de que cuando Karel llega a la casilla (x, y) sus casillas adyacentes ya han sido calculadas, podemos comenzar a rellenar las casillas de abajo hacia arriba y de derecha a izquierda, comenzando por la casilla $(1, 1)$.

```
void rellena() {
        rellenaCasilla();
        while(frontIsClear) {
            move();
            rellenaCasilla();
        }
        backturn();
        while(frontIsClear)
            move();
        turnleft();
        if (frontIsClear) {
            move();
            turnleft();
            rellena();
        }
    }
```

La instrucción `rellena` se ejecuta por primera vez en el program cuando Karel se encuentra en la casilla $(1, 1)$ mirando hacia el norte.

Estrellas - Solución.

Para resolver este problema es necesario implementar un algoritmo de ordenamiento, por lo que el ordenamiento de burbuja encaja perfectamente en esta ocasión.

El ordenamiento burbuja hace n recorridos a lo largo de la lista. Comparando los números adyacentes e intercambia los que no están en orden. Cada recorrido a lo largo de la lista ubica el siguiente valor más grande en su lugar apropiado. En esencia, cada numero sube como una burbuja hasta el lugar al que pertenece.

Para implementar esta solución se debe dividir el problema en sub-problemas, puesto que es necesario primero tener una forma de comparar objetos.

Comparación.

Los objetos a comparar son ternas x, y, z. El criterio que siguen es que un objeto A es mayor que un objeto B si $A_x > B_x$, en caso de que sean iguales entonces tiene que cumplirse que $A_y > B_y$, en caso de que $A_x = B_x$ y $A_y = B_y$ tiene que cumplirse que $A_z > B_z$.

Es necesario comparar 2 números individuales. Para esto podemos hacerlo de la siguiente manera donde Karel inicia viendo hacia el este sobre el número de la izquierda.

```
define contar(n) {
    if(nextToABeeper) {
        pickbeeper();
        contar(succ(n));
    }
    else {
        iterate(n)
            putbeeper();
        move();
        comparar(n);
    }
}
define comparar(n) {
    if(!iszero(n) && nextToABeeper) {
        pickbeeper();
        comparar(pred(n));
        putbeeper();
    }
    else if(iszero(n) && nextToABeeper)     // Número derecho es mayor.
        turnright();
    else if(!iszero(n) && notNextToABeeper) // Número izquierdo es mayor.
```

```
        backturn();
    else                                    // Números iguales.
        turnleft();
}
```

Karel terminará viendo hacia el norte si ambos números son iguales, hacia el sur si el número de la derecha es mayor o hacia el oeste si el número de la izquierda es mayor.

A partir de la orientación nosotros podemos tomar la próxima decisión.

```
define compararColumna() {
    iterate(3) {
        if(facingEast)
            contar(0);
        if(facingNorth) {
            move();
            turnleft();
            move();
            backturn();
        }
    }
    if(facingWest) { // Columna izquierda es mayor
        turnleft();
        while(frontIsClear)
            move();
        turnright();
        iterate(3) {
            intercambiaNumeros();
            backturn();
            move();
            turnleft();
            move();
            turnleft();
        }
        turnleft();
        while(frontIsClear)
            move();
    }
    else { // Columna derecha es mayor
        while(frontIsClear)
            move();
    }
    turnleft();
}
```

Tras comparar 2 columnas adyacentes, en la derecha quedará la columna mayor y en la izquierda la columna menor. Karel siempre terminará hasta abajo en la

columna de la derecha mirando hacia el este.

El intercambio de 2 números adyacentes se logra con recursividad de la siguiente manera.

```
define intercambiaNumeros() {
    if(nextToABeeper) {
        pickbeeper();
        intercambiaNumeros();
        putbeeper();
    }
    else {
        move();
        backturn();
        intercambia();
        backturn();
        move();
    }
}
define intercambia() {
    if(nextToABeeper) {
        pickbeeper();
        intercambia();
        putbeeper();
    }
    else
        move();
}
```

Ordenamiento.

El ordenamiento es una tarea más sencilla que la de comparar. Es necesario saber primero cuántos números se ordenarán, por lo que se debe medir la distancia de la lista.

Para esto, Karel inicia en la esquina inferior izquierda mirando hacia el este.

```
define medir(n) {
    if(nextToABeeper && frontIsClear) {
        move();
        medir(succ(n));
    }
    else {
        if(frontIsBlocked) {
            backturn();
            while(frontIsClear)
```

```
                move();
            backturn();
            ordenar(succ(n));
        }
        else {
            backturn();
            while(frontIsClear)
                move();
            backturn();
            ordenar(n);
        }
    }
}
define ordenar(n) {
    if(!iszero(n)) {
        iterate(pred(n))
            compararColumna();
        backturn();
        while(frontIsClear)
            move();
        backturn();
        ordenar(pred(n));
    }
}
```

El valor n en la instrucción `ordenar` indica la cantidad de números menos 1 a ordenar en este recorrido, ya que al término del i-ésimo recorrido los i objetos más grandes ya se encontrarán ordenados. Es por esto que podemos parar de ordenar cuando el valor n de la instrucción es 0. Karel debe recorrer la lista de izquierda a derecha, por lo que debe regresar al inicio de la lista al término de cada recorrido.