Córdoba, 15 de Junio del 2021

Proyecto Evaluatorio

Algoritmos y Estructuras de Datos II - TALLER

Backtracking y Programación Dinámica

Ejercicio 1: *Change-Making Problem*

El *problema de cambio de monedas* (*change-making problem* en inglés) consiste en buscar la mínima cantidad de monedas que permiten pagar cierto monto de dinero, teniendo a disposición una gama de denominaciones e infinitos ejemplares de cada una.

Lo llamaremos a veces “problema de la moneda” pero no se lo debe confundir con el *problema de la moneda de Frobenius* ni con el denominado *coin-change problem*, aunque tienen cierta relación.

En este ejercicio trabajaremos con la técnica de *backtracking* para encontrar la solución óptima. Necesitaremos interactuar con cantidades infinitas (culpa de que es el neutro del mínimo) por lo que vamos a abstraer los valores para representar la cantidad de monedas con el tipo `amount_t` (definido en **`amount.h`**) que tiene la siguiente interfaz:

Función	Descripción
`amount_t amount_inf()`	Devuelve la representación del valor infinito
`bool amount_is_inf(amount_t a)`	Indica si la cantidad en `a` es infinita
`bool amount_le(amount_t a1, amount_t a2)`	Indica si `a1` es menor o igual a `a2`
`bool amount_lt(amount_t a1, amount_t a2)`	Indica si `a1` es menor estricto a `a2`
`amount_t amount_sum(amount_t a1, amount_t a2)`	Calcula la suma entre `a1` y `a2`
`void amount_dump(amount_t amount)`	Muestra el valor de `amount` por pantalla

Para entender un poco más este tipo pueden analizar las siguientes propiedades:

```
amount_is_inf(amount_inf()) == true
amount_le(a, amoun_inf())   == true                ∀ a ∈ amount_t
amount_is_inf(amount_sum(a, amount_inf())) == true ∀ a ∈ amount_t
```

Se incluyen además los tipos `coin_t`, `currency_t` definidos en **`currency.h`** para los valores de las denominaciones de monedas y el monto de dinero a pagar. Para manejo de conjuntos se da El TAD *Set* definido en **`set.h`** con la siguiente interfaz:

Función	Descripción
`set set_empty(void)`	Construye un nuevo conjunto vacío
`set set_add(set s, set_elem e)`	Agrega el elemento `e` al conjunto `s`
`set set_clone(set s)`	Genera una copia de `s` en nueva memoria
`unsigned int set_cardinal(set s)`	Devuelve la cantidad de elementos que hay en `s`
`bool set_is_empty(set s)`	Indica si el conjunto `s` es vacío
`bool set_member(set_elem e, set s)`	Indica si el elemento `e` está dentro del conjunto `s`
`set set_elim(set s, set_elem e)`	Quita el elemento `e` del conjunto `s`
`set_elem set_get(set s)`	Devuelve un elemento del conjunto `s`
`void set_dump(set s)`	Muestra el contenido de `s` por la pantalla
`set set_destroy(set s)`	Destruye la instancia `s` liberando los recursos usados

Como los elementos de *Set* se abstraen con el tipo `set_elem`, las operaciones que el TAD necesita aplicar a estos elementos están especificadas en **`set_elem.h`** e implementadas en **`set_elem.c`**.

a) Completar la definición del tipo `amount_t` en **`amount.h`** y su implementación en el archivo **`amount.c`** asegurándose de satisfacer las propiedades descriptas y el correcto manejo para la representación del valor ∞ (infinito).

b) Completar del TAD Set la implementación de `set_clone()` que debe generar una copia de una instancia del tipo `set` en nueva memoria.

c) Implementar la función `change_making()` en el archivo **`change_making.c`:**

```
amount_t change_making(currency_t charge, set coins) {
    (...)
}
```

que dado un monto a pagar `charge` y un conjunto de denominaciones de monedas `coins` encuentra la solución del *problema de la moneda* utilizando la técnica de *backtracking*. Deben basarse en la solución mostrada en el material estudiado en el teórico-práctico.

⚠ **RECORDAR:** En ningún caso pueden modificar las interfaces de las funciones

Ejercicio 2: Programación Dinámica

Deben volver a resolver el problema de la moneda pero esta vez utilizando la técnica de *programación dinámica*. La función a implementar es:

```
amount_t
change_making_dyn(currency_t charge, coin_t *coins, unsigned int n_coins) {
    (...)
}
```

que toma el monto a pagar `charge`, un arreglo de denominaciones de monedas `coins` con `n_coins` denominaciones. La implementación debe realizarse en el archivo **`change_making_dyn.c`**. Se recomienda modularizar creando funciones locales para asistir en la creación y destrucción de la tabla dinámica. Nuevamente deben basarse en el algoritmo visto en el teórico-práctico.

RECORDAR: La matriz utilizada se debe crear dinámicamente (en el HEAP)

Testing Ej1 y Ej2

Se provee en la carpeta **`testing-changemaking`** un programa para comprobar el correcto funcionamiento de las implementaciones programadas, tanto en la versión con *backtracking* como para la que utiliza *programación dinámica*. El programa carga distintas configuraciones de experimentos desde archivos con el siguiente formato:

```
# CHARGE: 75 COINS: 8
1
5
10
```

La primera línea contiene el monto a pagar (que en el ejemplo serían $75), y de manera opcional se puede colocar el la cantidad de monedas esperada para la solución, que en este caso serían 8 monedas (10 x 7 + 5 x 1). Las siguientes líneas contienen las denominaciones disponibles.

Si la primera línea contiene el valor esperado (8), y se le ingresa al programa la opción **`-t`**, se verificará que el resultado de las funciones coincida con el mismo. Las opciones del programa son las siguientes:

Parámetro	Descripción
`-b`	habilitar el algoritmo de backtracking
`-d`	habilitar el algoritmo de programación dinámica
`-c <arg>`	Forzar el monto a pagar a "arg" (ignorando campo CHANGE de la primera línea del archivo)
`-t`	Habilitar las verificaciones (lee COINS desde primera línea)
`-f <files>`	Permite especificar en “files” la lista de archivos para los cuáles se quiere correr los algoritmos habilitados. Si no se incluye esta opción se pide la lista de denominaciones desde el teclado.

Como se sabe el problema de la moneda no siempre tiene solución, en cuyo caso el resultado sería infinito. Esta situación se puede representar usando el **`#`** para indicar el valor infinito. Por ejemplo:

```
# CHARGE:75 COINS: #
2
```

Cuando el programa realice el chequeo sabrá que el resultado de la función tiene que ser `amount_inf()`. La implementación del programa se encuentra en **`main.c`** y se compila desde esa carpeta ejecutando:

```
$ make
```

Y un ejemplo de prueba sería ejecutando:

```
$ ./change_making input/example.in
```

Ejercicio 3: Knapsack problem

El problema de la mochila consiste en cargar una mochila, que puede soportar un peso máximo, con los items de un catálogo de tal manera de maximizar el precio total de los artículos seleccionados, sin romper la mochila (ni lastimar a la persona que la carga).

Tributo a los trabajadores y trabajadoras de cadetería que se exponen para que nosotros nos cuidemos

Los algoritmos que solucionan el problema, por lo general computan el valor máximo al que puede llegarse, pero no indica qué elementos se seleccionaron. Por otro lado, tampoco hay una representación clara de “la mochila” (esto no es un defecto necesariamente, ya que el interés en realidad está puesto en encontrar la solución y no simular un experimento).

Se da en **`solve_knapsack.c`** un algoritmo que resuelve el problema de la mochila usando *backtracking* asistido por un TAD Mochila que deberán completar:

(*Notar que el kickstart tendrá más funciones que las listadas debajo*)

Función	Descripción
`knapsack knapsack_empty(unsigned int capacity)`	Construye una mochila vacía con capacidad `capacity`
`knapsack knapsack_clone(knapsack k)`	Crea una copia de `k` en nueva memoria
`knapsack knapsack_pack(knapsack k, item_t item)`	Agrega a `item` dentro de la mochila
`bool knapsack_greater_value(knapsack k1, knapsack k2)`	indica si los items de la mochila `k1` suman un valor total mayor que el de la mochila `k2`
`bool knapsack_can_hold(knapsack k, item_t item)`	Indica si la mochila tiene espacio suficiente para el objeto `item`.
`bool knapsack_is_full(knapsack k)`	Indica si la mochila está llena
`value_t knapsack_value(knapsack k)`	Devuelve la suma de los valores de los `items` que la mochila tiene dentro.
`void knapsack_dump(knapsack k)`	Muestra el contenido de la mochila
`knapsack knapsack_destroy(knapsack k)`	Destruye la mochila k

Van a notar también la presencia de un TAD Set que ha sido modificado para que sus elementos sean instancias de otro TAD (parecido a lo que sucedía en el ejercicio de diccionarios en el Proyecto 6). Entonces en el archivo **`set_elem.h`** ahora se agregan funciones para abstraer las operaciones necesarias para los elementos del conjunto en la implementación del TAD Set (comparar, destruir y clonar elementos).

Los objetos que se ingresan a la mochila se abstraen con el TAD Item que tiene las siguientes funciones:

Función	Descripción
`item_t item_create(string_t id, value_t value, weight_t weight)`	Construye un ítem a partir de un identificador, valor y peso.
`item_t item_clone(item_t item);`	Crea una copia de un ítem
`value_t item_value(item_t item);`	Devuelve el valor de un ítem
`weight_t item_weight(item_t item)`	Devuelve el peso de un ítem
`string_t item_id(item_t id);`	Devuelve el nombre de un ítem
`item_t * item_read_from_file(FILE *file, unsigned int *array_length)`	Lee un arreglo de items desde de un archivo y almacena la cantidad de ítems en `*array_length`
`item_t item_destroy(item_t item);`	Destruye un ítem

a) Deben completar en el archivo **`set.c`** la función **`set_clone()`** basándose en lo que ya habían hecho en el ejercicio *1a)*

b) Completar la implementación del TAD Knapsack en el archivo **`knapsack.c`**

Testing Ejercicio 3

La cátedra provee la implementación de la interfaz de línea de comandos, en el archivo **`main.c`**. El programa lee listas de *items* desde archivos de texto con el siguiente formato:

```
# KNAPSACK:15 VALUE:14
A:13:15
B:6:7
C:8:4
```

La primera línea contiene la capacidad máxima de la mochila (15), y de manera opcional se puede colocar el valor máximo esperado (14). Las siguientes líneas contienen los *items* (uno por línea) en el formato `nombre:valor:peso`.

Si la primera línea contiene el valor esperado (14), y se le ingresa al programa la opción **-t**, se verificará que el resultado de las funciones coincida con el mismo.

Parámetro	Descripción
`-b`	Habilitar el algoritmo de backtracking
`-w <arg>`	Forzar la capacidad de la mochila a "arg" (ignorando campo KNAPSACK de la primera línea del archivo)
`-t`	Habilitar las verificaciones (lee VALUE desde primera línea)
`-f <files>`	Lista de archivos para los cuáles se quiere correr los algoritmos habilitados. Si no se incluye esta opción se pide la lista de items desde el teclado en el formato `nombre:valor:peso.`

El programa se compila mediante

```
$ make
```

Algunos ejemplos para probar:

Bactracking:

```
$ ./knapsack -b -f input/example0.in
```

Backtracking verificando el valor obtenido:

```
$./knapsack -bt -f input/example*.in
```

Se deben corregir los *memory leaks* y los errores de acceso a memoria:

```
$ make valgrind
```

Criterios de Corrección

El que avisa no traiciona

Se evaluará del código presentado:

- Respetar los conceptos de **abstracción**, **ocultamiento**, **encapsulamiento** vistos a lo largo de la materia.
- Mantener buenas prácticas de programación como:
    - Correcta **indentación**
    - Buenos **nombres** para los elementos del código (variable, funciones, constantes)
    - Modularización para mantener el código lo más simple y legible posible
    - Uso de comentarios cuando sea necesario
    - Respeto de las interfaces
    - Chequeo de precondiciones y definición de invariantes de representación cuando lo amerite
- Correcto uso de la memoria, debiendo estar libre de *memory leaks* y de *invalid reads* e *invalid writes*
- Ser eficientes en las implementaciones en la medida de lo posible, sin resignar legibilidad del código ni extrema complejidad.
- Se correrá un software de detección automatizado de copias. En caso de detectar copias entre proyectos, se le solicitará a los alumnos involucrados una defensa oral con resolución de problemas en vivo. Si el alumno no aprobara esta defensa, se desaprobara (NA) el laboratorio de algoritmos II debiendo rendir en el final la parte laboratorio como “libre”.