Algoritmos y Estructuras de Datos II

Laboratorio - 05/06/2025

Laboratorio 6: Programación dinámica

`- Revisión 2025: Franco Luque`

Código

lab06-kickstart.tar.gz

Recursos

Recursos generales:
- Videos del Laboratorio en el aula virtual
- Documentación en el aula virtual
- Estilo de codificación:
  - Guía de estilo para la programación en C
  - Consejos de Estilo de Programación en C

Recursos específicos:
- Teóricos:
  - Backtracking
  - Programación dinámica
- Prácticos:
  - Práctico 3.3
  - Práctico 3.4
- Resoluciones:
  - Resoluciones de ejercicios varios 1
    - Problema de la mochila con Programación Dinámica
  - Resoluciones de ejercicios varios 2
- Lenguaje C:
  - Macros (para `MAX` y `MIN`)
  - Constantes `INT_MAX` y `INT_MIN` en `limits.h`.

Ejercicio 1: Problema de la moneda

Considerar el problema de la moneda visto en el teórico práctico:

Problema de la moneda

- Sean $d_1, d_2, \ldots, d_n$ las denominaciones de las monedas (todas mayores que 0),
- no se asume que estén ordenadas,
- se dispone de una cantidad infinita de monedas de cada denominación,
- se desea pagar un monto k de manera exacta,
- utilizando el **menor número de monedas posibles**.

a) Implementar la función `change()` que resuelve el "problema de la moneda" usando **programación dinámica**. Compilar y testear con:

```
$ gcc -Wall -Wextra -pedantic -std=c99 change.c tests.c -o tests
$ ./tests
```

b) En `tests.c` se provee un caso de test igual al ejemplo dado en el teórico. **Agregar al menos 5 nuevos casos de test**. Pensar tests de casos base y casos borde que puedan ser de ayuda para el debugging del algoritmo.

c) Se provee la función `print_table()` que puede ser útil para debugging. **Usar esta función al menos una vez para imprimir la tabla final del ejemplo dado en el teórico**. El resultado debe ser el mismo que se ve en el teórico:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	0	∞	∞	∞	1	∞	∞	∞	2	∞	∞	∞	3	∞	∞	∞	4
2	0	∞	1	∞	1	∞	2	∞	2	∞	3	∞	3	∞	4	∞	4
3	0	∞	1	∞	1	∞	2	1	2	2	3	2	3	3	2	3	3

Ejercicio 2: Problema de la mochila

Considerar el problema de la mochila visto en el teórico práctico:

Problema de la mochila

- Tenemos una mochila de capacidad W.
- Tenemos n objetos **no fraccionables** de valor $v_1, v_2, \ldots, v_n$ y peso $w_1, w_2, \ldots, w_n$.
- Se quiere encontrar la mejor selección de objetos para llevar en la mochila.
- Por mejor selección se entiende aquélla que totaliza **el mayor valor posible** sin que su peso exceda la capacidad W de la mochila.

a) Implementar la función `knapsack()` que resuelve el “problema de la mochila” usando **programación dinámica**. Compilar y testear con:

```
$ gcc -Wall -Wextra -pedantic -std=c99 knapsack.c tests.c -o tests
$ ./tests
```

b) En `tests.c` se provee un caso de test igual al ejemplo dado en el teórico. **Agregar al menos 5 nuevos casos de test**. Pensar tests de casos base y casos borde que puedan ser de ayuda para el debugging del algoritmo.

c) Se provee la función `print_table()` que puede ser útil para debugging. **Usar esta función al menos una vez para imprimir la tabla final del ejemplo dado en el teórico**. El resultado debe ser el mismo que se ve en el teórico:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	3	3	3	3	3	3	3	3	3
2	0	0	0	0	0	2	2	2	3	3	3	3	3	5	5	5	5
3	0	0	0	0	0	2	2	3	3	3	3	3	5	5	5	6	6
4	0	0	0	2	2	2	2	3	4	4	5	5	5	5	5	7	7

Ejercicio 3: Problema de la panadería

Considerar el problema de la panadería del práctico 3:

Una panadería recibe n pedidos por importes $m_1, \ldots, m_n$, pero sólo queda en depósito una cantidad H de harina en buen estado. Sabiendo que los pedidos requieren una cantidad $h_1, \ldots, h_n$ de harina (respectivamente), determinar el máximo importe que es posible obtener con la harina disponible.

a) Implementar un módulo con una función que resuelva el problema usando **programación dinámica**.

b) Implementar tests con **al menos 5 casos de test**. Pensar tests de casos base y casos borde que puedan ser de ayuda para el debugging del algoritmo.

Ítem eliminado (no hacer!): ~~**c)** Considerar la siguiente solución con **backtracking**~~:

~~*maxima_ganancia(i, h) = (i = 0 --> 0*~~
~~*| i > 0 & h < 0 --> - infinito*~~
~~*| i > 0 & h >= 0 -->*~~
~~*max(*~~
~~*maxima_ganancia(i-1, h), // no hago pedido i*~~
~~*maxima_ganancia(i-1, h - h_i) + m_i // hago pedido i*~~
~~*)*~~
~~*)*~~

~~Esta solución usa el *"- infinito"* para descartar automáticamente todos los casos en los que la harina no es suficiente. Implementar un algoritmo de **programación dinámica** basado en esta solución.~~

Ejercicio 4: Problema de la fábrica de automóviles

Considerar el problema de la fábrica de automóviles del práctico 3:

> Una fábrica de automóviles tiene dos líneas de ensamblaje y cada línea tiene n estaciones de trabajo, $S_{1,1}, \ldots, S_{1,n}$ para la primera y $S_{2,1}, \ldots, S_{2,n}$ para la segunda. Dos estaciones $S_{1,i}$ y $S_{2,i}$ (para $i = 1, \ldots, n$), hacen el mismo trabajo, pero lo hacen con costos $a_{1,i}$ y $a_{2,i}$ respectivamente, que pueden ser diferentes. Para fabricar un auto debemos pasar por n estaciones de trabajo $S_{i_1,1}, S_{i_2,2} \ldots, S_{i_n,n}$ no necesariamente todas de la misma línea de montaje ($i_k = 1, 2$). Si el automóvil está en la estación $S_{i,j}$, transferirlo a la otra línea de montaje (es decir continuar en $S_{i',j+1}$ con $i' \neq i$) cuesta $t_{i,j}$. Encontrar el costo mínimo de fabricar un automóvil usando ambas líneas.

a) Implementar un módulo con una función que resuelva el problema usando **programación dinámica**.

b) Implementar tests con **al menos 5 casos de test**. Pensar tests de casos base y casos borde que puedan ser de ayuda para el debugging del algoritmo.