Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2133 — Estructuras de Datos y Algoritmos 2'2023

Interrogación 2

8 de noviembre de 2023

Condiciones de entrega. Debe entregar solo 3 de las siguientes 4 preguntas.

Nota. Cada pregunta tiene 6 puntos (+1 punto base). La nota es el promedio de las 3 preguntas entregadas.

1. Backtracking

Registros Académicos de una nueva universidad necesita un sistema para asignar salas a los cursos luego de la toma de ramos. Considere que $S = \{s_1, \ldots, s_n\}$ y $C = \{c_1, \ldots, c_m\}$ son los conjuntos de salas y cursos, respectivamente. Luego, para una sala s_i su capacidad es $a(s_i)$ y para un curso c_j, su cantidad de inscritos es $\ell(c_j)$, su módulo horario es $1 \leq t(c_j) \leq 9$ y su día de la semana es $1 \leq d(c_j) \leq 5$. Una asignación es válida si cada sala tiene a lo más un curso en todo momento, todo curso c_j es asignado a alguna sala s_i en su horario y $0{,}8 \leq \ell(c_j)/a(s_i) \leq 1$, debido a que los estudiantes no quieren estar en salas demasiado grandes para su curso.

(a) Supongamos que todos los cursos tienen un solo módulo horario a la semana.

(i) [1 pto.] Defina variables, dominios y restricciones para el problema de asignar salas a cursos.

Solución.

Representaremos la asignación indicando en qué combinación sala-módulo-día se asigna un curso o si se deja vacío. Para esto, consideramos las variables

$$X = \{x_{t,d,i} \mid 1 \leq t \leq 9,\ 1 \leq d \leq 5,\ 1 \leq i \leq n\}$$

donde la variable $x_{t,d,i}$ indica qué curso se asigna en el módulo t del día d a la sala s_i. El dominio de estas variables es $\{0, 1, \ldots, m\}$, i.e. el indicador del curso que se asigna, incluyendo 0 para el caso en que la sala se deja vacía en ese momento.

Las restricciones son (1) si $x_{t,d,i} = j$, entonces debe cumplirse que $d(c_j) = d$ y $t(c_j) = t$, (2) si $x_{t,d,i} = j$, entonces $0{,}8 \leq \ell(c_j)/a(s_i) \leq 1$. La restricción de cantidad de cursos por sala en un momento dado está implícita en que las variables toman un solo valor del rango de ids de cursos.

Puntajes.
0.5 por definir variables y su dominio.
0.5 por especificar las restricciones que no están implícitas en las variables/dominios.
Observación: hay varias opciones para definir las variables, por ejemplo de la forma $x_{t,d,i,j}$ booleanas.

(ii) [3 ptos.] Proponga el pseudocódigo de un algoritmo de backtracking que retorne una asignación cursos válida o -1 si no existe.

Solución.

De acuerdo a las variables descritas, se propone un algoritmo que asigna las variables en un arreglo multidimensional de dimensión $9 \times 5 \times n$, de forma que $M[t][d][i]$ almacenará el valor de $x_{t,d,i}$. Primero, el algoritmo de backtracking que intenta completar M y responde si es posible hacerlo.

input : Arreglo multidimensional M, módulo t, día d, sala i
output: ¿Es posible asignar cursos a partir de la celda indicada?

```
Backtrack(M, t, d, i):
    if t, d, i fuera del rango :
        return true
    for j ∈ {0, 1, ..., m} :
        if Check(M, t, d, i, j) :
            M[t][d][i] ← j
            t', d', i' ← siguiente celda de M
            if Backtrack(M, t', d', i') :
                return true
    return false
```

input : Arreglo multidimensional M, módulo t, día d, sala i, curso j
output: ¿Se respetan las restricciones al asignar j en t, d, i?

```
Check(M, t, d, i, j):
    if 0,8 > ℓ(c_j)/a(s_i) OR ℓ(c_j)/a(s_i) > 1 :
        return false
    if t(c_j) ≠ t OR d(c_j) ≠ d :
        return false
    return true
```

Finalmente, el algoritmo que responde lo pedido utiliza el algoritmo definido anteriormente:

output: Asignación o -1

```
Solver():
    M ← arreglo multidimensional de 9 × 5 × n de ceros
    if Backtrack(M, 1, 1, 1) :
        return M
    return −1
```

Puntajes.
1.5 por definir el algoritmo de backtracking.
1.0 por definir el método que chequea restricciones.
0.5 por retornar lo pedido.
Observación: lo importante es la consistencia con la estructura escogida para almacenar la asignación. Además, basta con que el método `Check` explicite las restricciones necesarias en su solución. No es necesario encapsular todo como en `Solver`, pero sí se debe retornar asignación o -1.

(b) [1 pto.] Explique cómo modificar su solución de (a) para encontrar una asignación válida si es que los cursos pueden tener más de 1 módulo horario a la semana. No requiere mostrar pseudocódigo.

Solución.

Si consideramos que la sala debe ser la misma, existen dos opciones: (1) agregar una restricción adicional para verificar en `Check` o (2) propagar obligando que cualquier otra aparición del mismo curso solo pueda tomar un valor de sala.

Si consideramos que no hay restricción de sala, se puede extender el mismo algoritmo propuesto de manera que en lugar de m cursos, esta cantidad se aumenta viendo cada curso-módulo-día como un curso diferente. Luego, el algoritmo propuesto resuelve este problema sin necesitar más cambios.

Puntajes.
1.0 por explicar los cambios necesarios, si es que los hay.
Observación: en esta solución se muestran los dos posibles enfoques para abordar la pregunta. Ambos se consideran correctos dado que no se especifica restricción de sala en el enunciado.

(c) [1 pto.] Explique cómo modificar su solución de (a) si cada curso c_j tiene una prioridad diferente $p(c_j)$ y se exige que primero se asignen los cursos con mayor prioridad. No requiere mostrar pseudocódigo.

Solución.

Al momento de iterar sobre los valores de j posibles, este **for** debe recorrer los valores de j según la prioridad. Basta con ordenar los índices de acuerdo a $p(c_j)$.

Puntajes.
1.0 por explicar los cambios necesarios.

2. Programación dinámica

Dado un conjunto de enteros positivos $A = \{a_1, \ldots, a_n\}$, el problema `DivSum` consiste en responder si es posible construir dos conjuntos B_1, B_2 disjuntos, no vacíos, tales que $B_1 \cup B_2 = A$ y tales que tienen la misma suma de sus elementos. `DivSum` puede resolverse como problema de decisión, definiendo

$$p(k, S) := \quad \begin{array}{l} 1 \text{ si y solo si es posible construir un conjunto con} \\ \text{elementos de } a_k, \ldots, a_n \text{ con suma exactamente } S. \end{array}$$

y su relación recursiva

$$p(k, S) \;=\; p(k+1, S) \texttt{ OR } p(k+1, S - a_k)$$

donde `OR` es el operador de disyunción que evalúa primero el lado izquierdo. Si dicho lado es 1, entrega 1 sin evaluar el lado derecho. Esto significa que es posible que ciertos llamados de $p(k, S)$ no se realicen.

(a) [1 pto.] Identifique los casos base para $p(k, S)$. Defina el valor adecuado de $p(k, S)$ en tales casos.

Solución.

Como nos interesa que la recursión no persista, debemos considerar: (1) $k > n$ que representa el caso en que no quedan elementos, (2) $T < 0$ que representa suma negativa y (2) $T = 0$ que representa la condición de éxito (agotamos la suma buscada). Con esto, los casos se resumen en

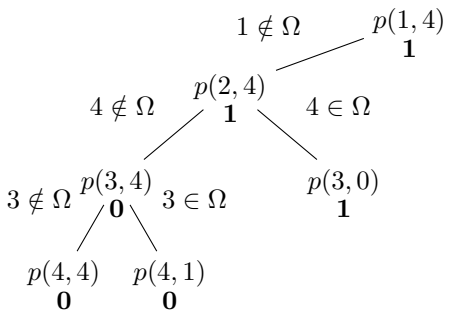
$$p(k, S) = \begin{cases} 0 & k > n \vee S < 0 \\ 1 & S = 0 \end{cases}$$

Puntajes.
0.5 por indicar los casos desfavorables.
0.5 por indicar el caso positivo.
Observación: pueden considerarse casos diferentes, mientras sean consistentes con (b). Es importante que los casos base sean binarios, porque $p(k, S)$ solo puede tomar valores 0/1.

(b) [2 ptos.] Resuelva el problema `DivSum` para $A = \{1, 4, 3\}$ mediante la definición de $p(k, S)$, usando un llamado inicial adecuado. Muestre el árbol recursivo de llamados que se genera, indicando qué elementos se incluyen/descartan en cada paso y mostrando cuándo la recursión llega a un caso base.

Solución.

El llamado que responde al problema `DivSum` para $A = \{1, 4, 3\}$ es $p(1, 4)$, donde el segundo argumento es la mitad de la suma de todos los elementos de A. Si $p(1, 4) = 1$, significa que es posible construir conjuntos con igual suma. Sea Ω uno de los subconjuntos. A continuación mostramos los llamados recursivos que indican cómo construir Ω.



Observamos que todo el subárbol derecho no se genera porque se llegó a un 1 en el lado izquierdo. Del árbol, deducimos que sí el problema `DivSum` tiene respuesta positiva para A. Más aún, el diagrama muestra $\Omega = \{4\}$ como uno de los conjuntos.

Puntajes.
1.0 por construir el árbol de llamados recursivos de acuerdo a la relación de recurrencia entregada en el enunciado.
0.5 por indicar valores descartados/incluidos en cada llamado.
0.5 por responder al problema para A.
Observación: En esta solución se asumió el orden arbitrario $a_1 = 1, a_2 = 4, a_3 = 3$ para los elementos de A. Cualquier orden es correcto. Tal como se indicó en (a), si se usaron casos base diferentes, está correcto mientras haya consistencia.

(c) [3 ptos.] Proponga el pseudocódigo de un algoritmo recursivo que para cualquier conjunto A de enteros positivos, responda el problema `DivSum` para A.

Solución.

Se asume M un arreglo bidimensional con celdas vacías. Primero proponemos el algoritmo de programación dinámica que calcula $p(k, S)$.

```
Dynamic(k, S):
    if k > n ∨ S < 0 :
        return 0
    if S = 0 :
        return 1
    else:
        if M[k][S] ≠ ∅ :
            return M[k][S]
        else:
            M[k][S] ← Dynamic(k + 1, S) OR Dynamic(k + 1, S − a_k)
            return M[k][S]
```

Luego, el algoritmo que resuelve el problema `DivSum` es

```
DivSum(A):
    S ← SUM(A)
    if S es impar :
        return 0
    return Dynamic(1, S/2)
```

Puntajes.
2.0 por algoritmo recursivo para $p(k, S)$.
1.0 por resolver el problema `DivSum` tomando la mitad de la suma.
Observación: No es necesario considerar como caso separado cuando S es impar al sumar los elementos de A. El problema también se resuelve correctamente solo llamando a `Dynamic`$(1, S/2)$.

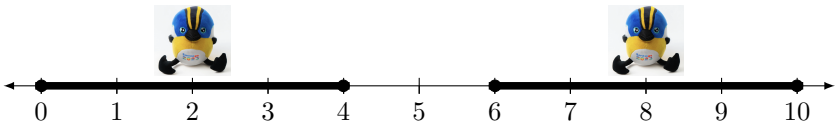
3. Algoritmos codiciosos

Para satisfacer la demanda de peluches de Fiu, la mascota de los Juegos Panamericanos y Parapanamericanos, se debe construir tiendas cerca de los recintos deportivos. Consideremos el problema de minimizar la cantidad de tiendas, para lo cual, representamos los recintos como puntos $R = \{r_1, \ldots, r_n\}$ en una misma recta, diciendo que un recinto está cubierto si en la recta hay una tienda a lo más a L kilómetros de él. Por ejemplo, si $L = 1$ y $R = \{0, 2\}$, la solución óptima es ubicar una tienda entre ambos recintos:

(a) [2 ptos.] Se propone la siguiente estrategia para escoger dónde colocar tiendas: "*ubicar la siguiente tienda en la posición que maximiza el número de recintos nuevos cubiertos y que no habían sido cubiertos por tiendas añadidas antes*". Demuestre que esta estrategia no es óptima en todos los casos.

Solución.

Para construir el contraejemplo, orientaremos el análisis buscando un ejemplo en el que la estrategia entregue 3 tiendas como "óptimo", cuando el verdadero óptimo sea 2. Para motivar la idea, consideremos las posiciones de 2 tiendas que no solapan sus zonas cubiertas, por ej, para $L = 2$ podríamos tener el siguiente escenario



Para forzar que la estrategia propuesta se equivoque, pondremos recintos tan cerca que puedan ser cubiertos por una tercera tienda entre las 2 originales. Primero ubicaremos 2 recintos extremos en 0 y 10 (para que estén en los extremos de nuestras tiendas óptimas), y segundo, ubicaremos 4 recintos muy cerca en los extremos interiores de las tiendas óptimas: en 3,4,6,7. Con esto, como hay 4 trecintos que pueden ser cubiertos con una tienda adicional en 5. La estretegia propuesta en el enunciado ubica su primera tienda en 5 y luego dos tiendas en 0 y 10, totalizando 3. Con todo esto, el contraejemplo es entonces $L = 2$ y $R = \{0, 3, 4, 6, 7, 10\}$.

Puntajes.
1.0 por contraejemplo.
1.0 por mostrar que en ese caso el óptimo no coincide con lo obtenido.
Observación: Se da 1.0 punto en total en caso que no se haya concretado un contraejemplo, pero haya intuición.

(b) [2 ptos.] Considere la siguiente estrategia codiciosa: "*con los recintos en orden creciente, si el primer recinto no cubierto es r, entonces ubicar la siguiente tienda a distancia L hacia adelante de r*". Proponga el pseudocódigo de un algoritmo codicioso que use esta estrategia para retornar la lista más corta con las ubicaciones de las tiendas que cubren el conjunto de recintos R con distancia L. Asuma que R es un arreglo no necesariamente ordenado.

Solución.

Se define el siguiente algoritmo para construir la lista de tiendas (números que indican su posición).

```
Greedy(R, L):
1    T ← lista vacía
2    R ← InsertionSort(R)
3    Insertar al final de T el dato R[0] + L
4    i ← 1
5    while i < n :
6        if T.last + L < R[i] :
7            Insertar al final de T el dato R[i] + L
8        i + 1
9    return T
```

Puntajes.
0.5 por ordenar secuencia de recintos.
1.5 por avanzar correctamente verificando rango.
Observación: Se puede usar cualquier algoritmo de ordenación.

(c) Si R contiene n recintos y se entrega como arreglo no necesariamente ordenado.

(i) [1 pto.] Determine si existe distinción entre mejor y peor caso para su algoritmo de (b). Justifique.
Solución.

El algoritmo propuesto solo depende de los mejores/peores casos del algoritmo de ordenación empleado. En este caso, el mejor caso ocurre cuando R está ordenado y el peor, cuando hay desorden en una cantidad lineal de elementos.

Puntajes.
1.0 por indicar de qué dependen los casos en su algoritmo propio.

(ii) [1 pto.] Determine la complejidad de los casos identificados en (i).

Solución.

La complejidad del loop es $\mathcal{O}(n)$ siempre. Luego, como `InsertionSort` es $\mathcal{O}(n)$ en el mejor caso, el mejor caso es lineal. El peor caso es cuadrático.

Puntajes.
1.0 por indicar los casos que corresponda.

4. Tablas de hash y orden lineal

Un sistema de detección de amenazas analiza la carga de los paquetes de datos que circulan buscando "firmas" que permitan identificar contenido malicioso. El patrón de firma es un número del tipo `0xF34C...0A`, con k dígitos hexadecimales. El sistema usa una tabla de hash `T` con encadenamiento para almacenar las firmas conocidas. Adicionalmente, a cada firma se asocia la cantidad de veces que es detectada y su registro en la CVE (*Common Vulnerabilities and Exposures*), el cual es un identificador único que sigue el formato: `CVE-YYYY-NNNN`, donde `YYYY` es el año de la asignación y `NNNN` es un número secuencial. Asuma que la tabla `T` tiene n amenazas almacenadas y la función de hash f es conocida.

(a) Proponga los algoritmos en pseudocódigo para los siguientes métodos:

(i) [0.5 ptos.] `agregaAmenaza(T,firma,cve)`, que recibe como entrada la tabla `T`, la firma de la amenaza, el identificador CVE de la amenaza e inicializa la cuenta de veces que ha sido detectada.

Solución.

`agregaAmenaza`$(T, firma, cve)$**:**
1 $c \leftarrow$ objeto nuevo
2 c.`firma` $\leftarrow firma$
3 c.`cve` $\leftarrow cve$
4 c.`count` $\leftarrow 1$
5 Insertar c en $T[f(firma)]$

Puntajes.
0.5 por insertar los datos en la lista de la celda $f(firma)$.
Observación: El conteo se puede iniciar en 0. También se pueden usar tuplas/arreglos en lugar de objetos para representar los elementos de las listas.

(ii) [0.5 ptos.] `esAmenaza(T,carga)`, que retorna el CVE asociado si la firma `carga` está registrada como amenaza aumentando su cuenta de detección en 1, o retorna -1 en caso contrario.

Solución.

`esAmenaza`$(T, carga)$**:**
1 **for** c *elemento de la lista* $T[f(carga)]$ **:**
2 **if** $c.firma = carga$ **:**
3 c.`count` $\leftarrow c$.`count` $+ 1$
4 **return** $c.cve$
5 **return** -1

Puntajes.
0.5 por chequear en la lista $T[f(carga)]$ y retornar acorde.
Observación: En esta solución, **for** se refiere a recorrer los elementos de la lista en cuestión.

(iii) [1 pto.] `detecciones(T)` que retorne, en tiempo lineal $\mathcal{O}(n)$, un arreglo A que contenga las CVE registradas en `T` y el número de detecciones observadas para cada una, ordenado por CVE. Puede asumir que el factor de carga de T es menor que 1 siempre.

Solución.

Para el siguiente algoritmo, se utiliza `RadixSort` ordenando un arreglo de tuplas de acuerdo al valor de la primera componente de estas (el CVE de amenazas).

```
   detecciones(T):
1      A ← arreglo vacío de tamaño n
2      for i ∈ {1, ..., m} :
3          for c elemento de la lista T[i] :
4              Insertar (c.cve, c.count) al final de A
5      A ← RadixSort(A)
6      return A
```

Puntajes.
0.5 por revisar todas las amenazas registradas.
0.5 por ordenar cumpliendo con la complejidad solicitada.
Observación: Tal como en esta solución, basta describir las modificaciones necesarias a algoritmos de ordenación vistos en clase para poder usarlos. Es importante notar que la línea 4 se ejecuta un total de n veces.

(b) [2 ptos.] Se pide proponer un algoritmo `top(T)` que retorne en tiempo $\mathcal{O}(n)$, un arreglo P que contenga la lista de las CVE registradas como amenaza en `T` ordenado de mayor a menor por el número de detecciones. ¿Es posible cumplir con lo solicitado? Justifique su respuesta.

Solución.

Es posible entregar tal algoritmo. Usando `detecciones` del inciso anterior, se obtiene un arreglo de pares (`cve,count`). Usando `RadixSort` sobre este arreglo, ordenando según el valor de la segunda componente, se obtiene el orden pedido. Ahora bien, cabe notar que el tiempo de este algoritmo será $\mathcal{O}(n)$ siempre que el número de dígitos k necesarios para representar el número de detecciones cumpla $k \in \mathcal{O}(n)$. Si no es así (es decir, han ocurrido muchas más detecciones que registros de amenazas nuevas), entonces el algoritmo no es lineal.

Puntajes.
1.0 por explicar cómo obtener el arreglo pedido.
1.0 por argumentar que su complejidad cumple.

(c) [2 ptos.] Proponga un algoritmo en pseudocódigo para el método `count(T,cve)` que permite obtener en tiempo $\mathcal{O}(1)$ el número de detecciones registrados para un CVE dado. *Sugerencia*: modifique `agregaAmenaza` de la parte (a) de modo de contar con una tabla de hash auxiliar `C` con encadenamiento que use como llave el código CVE y lo relacione con la firma y cuenta de detecciones registrada en `T`.

Solución.

Modificamos primero el método del inciso (a), para que se actualice al mismo tiempo la tabla auxiliar `C`. Asumimos que las funciones de hash de ambas tablas son conocidas, a saber, f_1, f_2.

```
   agregaAmenaza(T, firma, cve):
1      c_1 ← objeto nuevo
2      c_1.firma ← firma
3      c_1.cve ← cve
4      c_1.count ← 1
5      c_2 ← objeto nuevo
6      c_2.cve ← cve
7      c_2.pointer ← puntero a c_1
8      Insertar c_1 en T[f_1(firma)]
9      Insertar c_2 en C[f_2(cve)]
```

Con esto, el algoritmo que encuentra el conteo dado un CVE es

```
count(T, cve):
1     for c elemento de la lista C[f_2(cve)] :
2         if c.cve = cve :
3             return c.pointer.count
4     return −1
```

Puntajes.
1.0 por conectar elementos de ambas tablas.
1.0 por entregar el valor pedido sin buscar en T.