Ayudantía I2

Hashing
Técnicas Algorítmicas
Grafos

Ejercicio 1
Tablas de Hash

El sistema del DCCentro médico usa la hora (hr) en formato hh:mm, para saber qué paciente tiene la hora y qué doctor lo atiende.

Actualmente, se está atendiendo desde las 8:00 (primera hora disponible) hasta las 17:00 (última hora disponible).

Se tiene el conjunto E de especialidades (fijas). Lamentablemente como no siempre están los mismos doctores en turno, no hay un registro fijo de los doctores disponibles.



Pregunta 1

Para organizar esta información, actualmente se está usando una tabla de hash donde h(hr): [00.00, 23:59]->[0,...,23]. h(hr):=hr.hh.
Si la casilla que se quiere usar está ocupada, se busca la siguiente que esté desocupada, si no se encuentra ninguna, se da la vuelta y parte desde el inicio hasta llegar a la posición original. Si todas las posiciones están ocupadas crea una nueva tabla de hash guardando (paciente, doctor).

Para buscar una cita se sabe la hora (hr) y la especialidad (e). Por lo que Search(hr, e) deberá buscar de manera secuencial en todas las tablas de hash que existan, por la existencia del paciente.

Pregunta 1

Es importante notar que (d,hr,paciente) es una tupla única, es decir, no pueden existir dos pacientes con la misma hora y el mismo doctor. Pero si pueden existir varios pacientes con una hora dentro de la especialidad.

a) ¿Cuál es el peor tiempo de búsqueda de un paciente? Exprese su resultado en función de n como la cantidad de elementos actualmente en la tabla.

Pregunta 1



Pregunta 1. a) - Solución

El peor tiempo de búsqueda será de **O(n)**.

Ya que, si queremos buscar un paciente que tiene una hora para la cual exista una colisión (donde más nos demoramos en buscar), como es sondeo lineal, que una hora nos lleve a un índice no garantiza que ese sea el paciente, por lo que debemos recorrer todo el resto de los elementos de manera lineal para comprobar si es que el paciente existe o no.

b) Si sabemos que el 80% de las horas que se piden están concentradas desde las 10:00 hasta las 15:00, **¿cómo se puede mejorar el tiempo promedio de búsqueda?**
Explique **qué cambios** le haría al esquema descrito y **por qué mejoran el rendimiento** del esquema. Si debe cambiar o agregar funciones de hash basta con explicar de manera clara la distribución de las llaves a través del espacio de índices.



Pregunta 1

Pregunta 1. b) - Solución

Cambios a la función h.

- h: [08.00, 17:00] -> [0, 9]

- Distribución de h: Como el 80% está en 5 horas (10-15), y el 20% en las horas restantes debemos construir una función h' que distribuya las llaves de que en el espacio de llegada [10-15 (80%)| 8-9:59 U 15:01-18 (20%)]

Horas dentro de [10:00, 15:00]								**[8:00, 10:00[**	]15:00, 18:00]
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	9

Pregunta 1. b) - Solución

Cambios al esquema

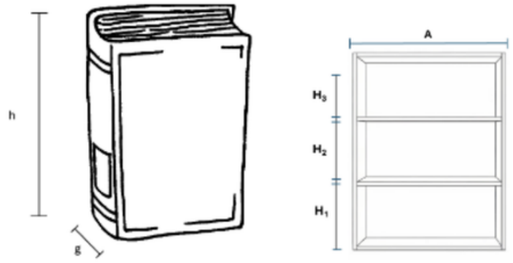
- Direccionamiento abierto -> Encadenamiento.
- Nueva hash: h_2(e): E -> [0,..,|E|-1].
- Encadenamiento 2° nivel:
  - En cada casilla A[h(hr)] tenemos una tabla de hash B, que ocupa h_2 para distribuir sus espacios. Es decir que para acceder a una cita a las hr de la especialidad e tenemos que hacer A[h(hr)][h_2(e)]

Ejercicio 2
Técnicas Algorítmicas

Bob acaba de comprar una repisa de libros de 3 pisos de diferentes alturas H_1, H_2, H_3 cms y de A cms de ancho y con una capacidad máxima de W kg. Los libros se deben acomodar en forma vertical, es decir, con su grosor ocupando espacio en el ancho de la repisa.

Además, cuenta con un conjunto de libros a ser acomodados en la repisa. Cada libro L_i se caracteriza por su altura h_i cms, su grosor g_i cms y su peso w_i kg.

Suponga que todos los libros caben de alto en alguna repisa y que cada piso de la repisa puede acomodar a varios libros.



Pregunta 3 - I2-2025-1

En los siguientes puntos, seleccione el uso de una estrategia algorítmica distinta (es decir, no puede usar la misma técnica en dos puntos distintos) entre Backtracking, Codiciosa o Programación Dinámica que mejor se ajuste para resolver cada uno de los siguientes problemas. En cada caso debe plantear las restricciones, la función objetivo, la ecuación de recurrencia o la estrategia codiciosa según corresponda, describir la solución y justificar la elección realizada.

a) (2 pts.) Asignar libros a pisos de la repisa de modo que se cumpla con todas las condiciones.

b) (2 pts.) Distribuir libros para maximizar el número total de libros ubicados. Sin exceder ancho ni peso soportado por la repisa.

c) (2 pts.) Ordenar libros para minimizar el espacio sobrante en cada piso sin exceder el peso ni ancho.

Pregunta 3 - I2-2025-1

Pregunta 3 I2-2025-1 - Solución

Restricciones generales:

- Restriccion de ancho por piso

$$\sum_{i=1}^{n} g_i <= A \text{ para cada } \textbf{piso}$$

y que al agregar un libro más se supere A

- Restriccion de peso en repisa completa

$$\sum_{i=1}^{n} w_i <= W \quad \text{para la repisa completa}$$

y que al agregar un libro más se supere W

- Resricción de altura por piso

$$h_i <= H_1, H_2, H_3$$

Pregunta 3 I2-2025-1 a) - Solución

Objetivo: encontrar una asignación libro-piso que cumpla ancho, peso y altura. No hay nada que optimizar.

Backtracking: al no haber función objetivo, no aplica greedy ni DP de forma natural. Se explora el espacio de asignaciones y se poda en cuanto se viola alguna restricción.

- Para cada Li, probar piso 1, 2, 3 (o "no colocar"), validando hi ≤ Hj, ancho acumulado + gi ≤ A, peso acumulado + wi ≤ W.
- Si falla alguna restricción → backtrack, probar la siguiente opción.
- La primera asignación completa que pasa todo es la solución (no se busca optimalidad).

Pregunta 3 I2-2025-1 b) - Solución

Objetivo: maximizar el número de libros sin exceder ancho por piso ni peso total.

Estrategia Greedy: ordenar por gi ascendente (o por densidad wi/gi) y colocar cada libro en el primer piso donde quepa.

Justificación: Maximizar cantidad (no valor) bajo una capacidad se resuelve naturalmente priorizando lo "más barato" en el recurso que se llena primero, mismo principio que en mochila. Además, la elección es razonable dado que a) y c) ya usaron las otras dos técnicas.

Pregunta 3 I2-2025-1 c) - Solución

Objetivo: para cada piso j, minimizar A - Σ gi (espacio sobrante), sin violar ancho, peso ni altura.

DP: equivale a 3 mochilas (una por piso), con la decisión ¿Li va al piso 1, 2, 3, o no se coloca? Son subproblemas que se repiten según el espacio/peso restante (estructura subóptima y problemas solapados)

El espacio disponible después del libro Li es d[i][j] por cada piso [j]. La ecuación de recurrencia es tiene 4 opciones: colocar el libro en Piso1, Piso2, Piso3 o no colocar el libro.

Ejercicio 3
Grafos - Modelación/DFS

1. Una forma de recorrer (y descubrir) un grafo no direccional $G = (V, E)$ es la siguiente: Primero, elegimos un vértice cualquiera de V y lo ponemos en una cola Q inicialmente vacía; Q es una cola común, del tipo "el primero que entra es el primero que sale". Luego, hacemos lo siguiente: Repetidamente, sacamos el primer vértice de Q, llamémoslo u; recorremos la lista de adyacencias de u y cada vértice que encontramos en la lista lo ponemos en Q.

Pregunta 4 - I2-2024-02

a) (0.5 pts.) Explica qué se puede hacer para que este algoritmo no "repita" vértices, es decir, para que los vértices que ya han sido descubiertos no vuelvan a ser descubiertos.

Pregunta 4 - I2-2024-02

Pregunta 4.a.i) - Solución

Basta con tener algo que nos indique si **lo marcamos o no**!

En DFS se usaban los colores para marcar cuando comenzamos la ejecución como booleanos: **blanco** significa que no he visitado aún, **gris** significa que recién visité dicho nodo y estoy en su ejecución, y **negro** significa que ya finalicé. En general, con **dos colores es suficiente** si quieren ver si ya se ejecutó con tal de no repetirlo (actuando como true or false en booleanos)

Una manera adicional para hacerlo es tener una **lista** que marque si fueron descubiertos, pero **no es lo ideal** ya que posee un gasto adicional de memoria y buscamos eficiencia :)

ii) ¿Cuál es la condición de término del algoritmo?

Pregunta 4 - I2-2024-02

Pregunta 4.a.ii) - Solución

La condición de término se va a dar cuando ya **no tengamos más nodos por ser descubiertos.**

Para que esto ocurra, se tiene que dar que no tengamos más vértices que sacar de la cola Q, ya que, junto al inciso anterior (no visitamos vertices ya descubiertos), no nos deberían quedar pendientes por visitar.

Es decir, nuestra condición de término del algoritmo es que la **cola Q se encuentre vacía**.

iii) ¿Qué información adicional puede extraerse de este algoritmo en términos de la distancia, medida en número de aristas, que hay entre el primer vértice que pusimos en Q y cada uno de los otros vértices que van siendo descubiertos?

Pregunta 4 - I2-2024-02

Pregunta 4.a.iii) - Solución

Para este inciso es importante fijarse en como se recorre cada uno de los caminos disponibles para llegar a un nodo en específico.

El algoritmo siempre va a descubrir un vértice utilizando **el menor número posible de aristas desde el vértice original**. Esto se da ya que la cola no va a visitar distancias más largas.

Esto se refiere a que, si partimos con un vertice v, comenzamos con **distancia 0**. Se agregan todos los **vecinos directos** a la cola Q, y todos ellos van a tener una **distancia de 1**. Una vez recorridos todos los de distancia 1, se agregaron todos los de **distancia 2** a la cola, y así consecutivamente.

iv) Describe una versión completa del algoritmo —es decir, incluyendo los tres puntos anteriores— en pseudocódigo similar al usado en clases para describir el algoritmo DFS.

Pregunta 4 - I2-2024-02

Pregunta 4.a.iv) - Solución

```
algoritmoPedido(s):   —s es el vértice (arbitrario) de partida
for each u in V–{s}:
        u.color ← white        —asignación inicial de color a todos los vértices
        u.δ ← ∞                —asignación inicial de distancia desde el primer vértice
s.color ← gray          —acabamos de "descubrir" el vértice de partida
s.δ ← 0                 —... que está a distancia 0 de sí mismo
Q ← cola        —inicializamos una cola Q
Q.enqueue(s)            —...y ponemos s en la cola
while !Q.empty( ):      —condición de término del algoritmo
        u ← Q.dequeue( )      —sacamos el primer vértice de la cola
        for each v in α[u]:   —recorremos la lista de vértices adyacentes a u
                if v.color = white:     —si el vértice adyacente v no ha sido descubierto
                        v.color ← gray          —... lo descubrimos
                        v.δ ← u.δ+1     —...calculamos su distancia al primer vértice
                        Q.enqueue(v) —...y lo ponemos en la cola
```

b) Queremos saber si estando en una estación del metro, puedo ir a cualquier otra estación usando únicamente el metro. Para esto, representamos la red del metro como un grafo direccional, de la siguiente manera: Cada estación es un vértice, y, si en una misma línea del metro la estación v es la próxima estación a la estación u, entonces (y sólo entonces) incluimos una arista direccional (u, v).

Pregunta 4 - I2-2024-02

i) Describe un algoritmo eficiente en pseudocódigo que permita responder la pregunta original; tu algoritmo puede hacer uso sólo del algoritmo DFS estudiado en clases.

Pregunta 4 - I2-2024-02

Pregunta 4.b.i) - Solución

Si representamos la red del metro como se describe arriba, entonces la respuesta a la pregunta inicial es “sí” si y sólo si el grafo es una sola gran componente fuertemente conexa.

Por lo tanto, el algoritmo que se pide es el algoritmo de componentes fuertemente conexas (CFCs) con la única particularidad de que al terminar hay que ver cuántas CFCs encontró: si encontró exactamente una, entonces el algoritmo imprime “sí”, en cualquier otros caso, “no”.

Cuando el enunciado dice “*tu algoritmo puede hacer uso sólo del algoritmo DFS estudiado en clases*,” quiere decir que hay que escribir el algoritmo con un nivel de detalle similar al usado en clases (que hace uso de DFS), y hay que agregarle la condición final; no se puede poner como respuesta, “*el algoritmo CFC estudiado en clases*.”

ii) Justifica que tu algoritmo es eficiente.

Pregunta 4 - I2-2024-02

Pregunta 4.b.ii) - Solución

El algoritmo básicamente hace dos recorridos **DFS** del mismo grafo, primero, sobre la versión original del grafo, y luego sobre la versión transpuesta. Luego, es $O(V+E)$, es decir, es lineal en el "tamaño" del grafo: números de vértices y número de aristas.

Ejercicio 4
Kruskal

La gente de la tierra de Omashu se toma los grupos de amigos muy en serio. Tan en serio, que podemos describirlos matemáticamente (son clases de equivalencia):

- Si a es amigo de b entonces b es amigo de a
- Cada persona es amiga de sí misma y cada persona pertenece a un solo grupo de amigos
- Si a forma parte del grupo X, y b es amigo de a, entonces necesariamente b forma parte de X.



Pregunta 2 - I3-2020-2

a) Dada una lista F de pares de forma (a, b) que indican amistad entre la persona a y la persona b, describe un algoritmo lineal en el número de pares que calcule la cantidad de grupos de amigos distintos que existen en Omashu.

Pregunta 2 - I3-2020-2



Pregunta 4a - Solución

Se puede apreciar que cada grupo de amigos corresponde a un conjunto disjunto, donde nunca va existir un amigo que este en dos grupos distintos. Para la resolución del problema, se cuenta con un grafo donde cada nodo es una persona y las aristas son los pares (a, b) de la lista F. Podemos notar que se pide calcular la cantidad de conjuntos disjuntos, los cuales se representan por los grupos de amigos, es por esto que se puede utilizar el algoritmo visto en clases Kruskal con modificaciones, tal como se muestra a continuación

Pregunta 4a - Solución

```
1:  procedure OMASHU(F)
2:      groups := 0
3:      for (a, b) ∈ F do
4:          if a = b then
5:              make_set(a)
6:              groups := groups + 1
7:          end if
8:      end for
9:      for (a, b) ∈ F do
10:         if find_set(a) ≠ find_set(b) then
11:             union(a, b)
12:             groups := groups − 1
13:         end if
14:     end for
15:     return groups
16: end procedure
```

Pregunta 4a - Solución

En primer lugar, como cada persona es amiga de si misma, se inicializa a cada nodo como su propio representante y junto con esto, se tiene un contador que se le va sumando uno por cada vez que se encuentra una nueva persona, de esta forma, después de haber recorrido todos los pares pertenecientes a F, el contador groups tiene el total de nodos que tiene el grafo, que inicialmente van a ser nuestros conjuntos disjuntos Luego, se vuelve a recorrer todos los pares de F y si se encuentra que el representante de a es distinto al de b, se une los conjuntos, ya que pertenecen al mismo grupo, dejando al mismo representante para ambos. Como se unen los conjuntos, disminuye la cantidad de conjuntos disjuntos, por lo que hay que restarle uno a groups Finalmente, después de haber recorrido todos los pares de F, se tiene la cantidad total de grupos de amigos distintos que existen en Omashu

b) Además de la lista F anterior, se te da una lista U con tríos de la forma (a, b, w). Cada trío indica que a y b no son amigos, pero podrían serlo si se les paga una cantidad positiva w de dinero. Describe un algoritmo a lo más linearítmico (es decir, del tipo nlogn) que calcule el costo mínimo necesario para que todos los habitantes de Omashu formen un solo gran grupo de amigos.

Pregunta 4 - IX-20XX-X

Pregunta 4b - Solución

Este es similar al anterior. Solo que ahora primero se unirán ambas listas antes de seguir con el resto del algoritmo. Ahora en vez de calcular el número de grupos, se irá sumando el costo cada vez que haya una unión, retornando finalmente el costo total. Este algoritmo entrega complejidad n log n dada la demostración vista en clases.

Pregunta 4b - Solución

```
1:  procedure OMASHU(F, U)
2:      for (a, b) ∈ F do
3:          union(U, (a, b, 0))
4:      end for
5:      sort(U) de menor a mayor w
6:      cost := 0
7:      for (a, b, w) ∈ U do
8:          if a = b then
9:              make_set(a)
10:         end if
11:     end for
12:     for (a, b, w) ∈ U do
13:         if find_set(a) ≠ find_set(b) then
14:             union(a, b)
15:             cost := cost + w
16:         end if
17:     end for
18:     return cost
19: end procedure
```

Feedback