Índice

A - Enunciado………………… 2

B - Estrategia de resolución….. 2

C - Análisis de orden………….. 4

D – Respuestas/Conclusiones.…. 5

E - Código fuente…………….… 7

A – Enunciado

Ustedes han sido contratados como casamenteros de una comunidad muy cerrada , formada por N hombres y N mujeres, todos solteros, y que deben casarse (todos) entre sí este año.

Basarán su asignación en las preferencias de cada uno (es decir que le pedirán a cada hombre que haga un ranking de las mujeres, calificándolas de 1 a N, y de igual modo le pedirán a cada mujer que haga un ranking de todos los hombres).

Para mantener la buena fama de casamenteros que tienen (que los ha hecho muy solicitados por diversas comunidades) deben evitar que los matrimonios se separen al poco tiempo. En particular quieren evitar armar parejas inestables ( decimos que estamos en una situación inestable si un hombre y una mujer que están casados con otros preferirían estar casados entre sí). El problema de armar las parejas sin que haya situaciones inestables se conoce como el problema de los matrimonios estables.

Se pide:

1) Resolver el problema por Backtracking.

a) Calcular el orden.

2) Programar el algoritmo de Gale-Shapley para resolver el problema.

a) Calcular el orden.

b) Justificar que la asignación es completa y estable, es decir que el algoritmo no deja personas solteras y la asignación realizada es estable.

c) Justificar que el algoritmo termina siempre.

d) ¿El algoritmo es simétrico? Justificar.

e) Si se les permitiera a las mujeres cambiar la lista de preferencias durante la ejecución del algoritmo, ¿Cómo podría hacer una mujer dada para conseguir el hombre óptimo? Explicar.

f) ¿De qué estrategia algorítmica se trata? Justificar.

B – Estrategia de resolución

En primer lugar, se eligió Java 2 como entorno de desarrollo, esencialmente debido a que facilita la tarea del programador, y el hecho de que el enfoque OO resulta natural y sencillo para este problema en particular.

En ese sentido, la clase Individuo representa a un integrante cualquiera de la comunidad, sea hombre o mujer. Los atributos de un Individuo son su nombre (necesario para armar el archivo de salida), su sexo , su estado civil y dos colecciones de Individuos. Dichas colecciones son colas de prioridades (de máximos) cuyos elementos son los individuos (del sexo opuesto, claro) que le interesan, y otra con aquellos que están interesados en él. En ambos casos, la cola de prioridad está implementada con la más eficiente estructura de datos para ese propósito, vale decir la parva (o montículo, o heap) de máximos. Dicha estructura permite obtener el máximo en tiempo constante, y agregar o buscar un nodo en tiempo logarítmico.

La clase Pareja es una agregación de dos individuos, un hombre y una mujer, que están comprometidos pero aún no casados. Ergo, pueden llegar a romper su compromiso.

Una asignación es una agregación de parejas, construida ya sea por Backtracking o con el algoritmo de Gale-Shapley, por lo cual al final del programa el resultado final, o sea la asignación pedida, es estable. Hasta ese momento, el carácter de los compromisos es transitorio. La clase encargada de efectuar las asignaciones es Agencia, que a ese efecto posee todos los datos necesarios sobre los individuos de la comunidad. Vale decir, sus preferencias y la lista de parejas que representa a la asignación.

Sintéticamente, el diagrama de clases es el siguiente:

tdatp2

Agencia

Individuo

ArchivoSalida

1 1 1 1 1 2N

- hombres:ArrayList

-mujeres:ArrayList

-parejas:ArrayList

-pretendidos: Parva

-pretendientes: Parva

+asignarConBT( )

+asignaConGS( )

1 1 2 1

ArchivoEntrada

1 1

parva

1

N 2

TablaHash

Pareja

Parva

1

1

N

Elemento

Para implementar la solución por Bactracking no hay muchas variantes, dado que algorítmicamente hablando el esquema básico es invariante:

extender (Var S:Solucion){

si (listo(S)) then return(S);

else (para cada hijo H de S){

si (factible(S,H)){

S:= registrar(S,H);

extender(S);

S:=olvidar(S,H);

}

}

}

Lo que varía para cada problema es:

i – Las estructuras de datos empleadas, que determinarán el recorrido a efectuar para “barrer” todos los hijos de S, y tendrán mucho peso en el orden del algoritmo.

ii – Las funciones listo(S), factible(S,H), registrar(S,H) y olvidar(S,H), las cuales dependen del problema considerado y las estructuras empleadas para su resolución.

En el problema tratado, se hizo necesario que el procedimiento extender recibiera una Agencia a, para que todas las recursividades conozcan los datos de los individuos de la comunidad. Además, primero se registra la extensión y luego se verifica si es factible. De ser ello así, se prosigue la extensión con el hombre siguiente al actual.

Para este caso, la solución S es una asignación (agregación de parejas), y los hijos son parejas. El recorrido para todos los hijos de S se efectúa de la siguiente manera: Si no está lista la asignación, prueba formar una pareja entre el hombre actual (que inicialmente es el primero de los N) con todas las mujeres. Tanto al salir de cada recursividad como en el caso de que la solución extendida no sea factible, se olvida la extensión.

En cuanto a la función listo(S), devuelve verdadero si y sólo si la asignación es completa y estable, vale decir que no hay ningún par bloqueante, y hay N parejas.

La función factible(S,H), renombrada como esEstable, controla que la asignación actual sea estable. Para ello, se procede por inducción: Al agregar la primera pareja, la asignación es estable. Cuando se adiciona otra, hay que verificar la estabilidad entre las dos. Cuando se agregue la tercera, las dos anteriores ya habrán sido verificadas entre sí, por lo que sólo habrá que chequear la tercera contra las otras dos. Inductivamente, cuando se inserte la j-ésima pareja, será suficiente compararla una a una con las parejas de 1 a j-1. La verificación de la estabilidad entre dos parejas P1 y P2 (función esBloqueante) consiste simplemente en constatar que no ocurra alguna de las siguientes situaciones: P1.h prefiere a P2.m sobre su pareja actual y a P2.m le ocurre lo mismo, o bien P1.m desearía estar con P2.h antes que con su compañero actual, al igual que P1.m.

En lo que respecta a las funciones registrar(Pareja nueva) y olvidar(), consisten sencillamente en agregar la pareja nueva y retirar la última pareja de la asignación, respectivamente.

En cuanto al manejo de los datos, éstos son obtenidos del archivo de texto de entrada (ver código para más información) y manejados con una tabla de hash. Esto es debido a que con esta tabla, es más eficiente acceder a un objeto Individuo a partir de su nombre (que es como viene en el archivo). Para detalles acerca de la función de hash, ver comentarios en el código fuente de la clase TablaHash.

C – Análisis de orden (órdenes teóricos)

“n” es la cantidad de parejas a formar.

Individuo:

|  |  |  |
| --- | --- | --- |
| **Función** | **Órdenes teóricos** | **Análisis** |
| Crear | O(1) |  |
| addPretendiente | O(log(n)) | Es la inserción en una parva |
| setCalificacion | O(n) | Debe asegurar el tamaño de un vector. |
| pretendienteFavorito | O(1) | Debe obtener el máximo de una parva sin eliminarlo. |
| olvidarPretendientes | O(n) | Vacía la parva |
| primerIndividuo | O(n) | Ordena los individuos por calificación. |
| nextIndividuo | O(1) | Obtiene el siguiente de una lista ordenada. |
| RESTO DE LOS MËTODOS | O(1) |  |

Parva: Si n es la cantidad máxima inicial de elementos en la parva (que siempre es conocida, porque equivale a la cantidad de hombres -mujeres- y se determina durante el parseo). Otra consideración importante es que nunca habrá más de n elementos en la parva, por lo que nunca habrá necesidad de expandir dinámicamente la estructura.

|  |  |
| --- | --- |
| **Función** | **Orden teórico (tiempo)** |
| Crear | O(n) |
| ¿Estructura vacía? | O(1) |
| Vaciar/Borrar | O(n) |
| Obtener Primero/Máximo (sin extraerlo) | O(1) |
| Obtener Primero/Máximo (extrayéndolo) | O(log n), porque hay que restablecer la propiedad heap luego de extraer el máximo. Como ello se hace bajando la nueva raíz (que era el último elemento)por el “árbol” , se harán, a lo sumo, tantos intercambios como niveles tenga el árbol (o sea log N, porque el árbol es balanceado). |
| Insertar elemento | O(log n). Quien baja por el “árbol” ahora es el nuevo elemento. |

Agencia:

|  |  |  |
| --- | --- | --- |
| **Función** | **Órdenes teóricos** | **Análisis** |
| Asignar con Backtracking |  |  |
| asignarConGS | O(n2)  (en el peor caso que hemos estimado, pero no puede ser mejor que O(n log2(n)). | En cada etapa, para cada uno de los hombres solteros (en el peor caso n), se elige la mujer favorita que aún no lo ha rechazado en O(log(n)) y se agrega como pretendiente de ella, en O(log(n)).  Luego, cada una de las n mujeres se queda con el preferido – O(log(n)- y se olvida de los demás – O(1)-.  Al final del algoritmo, se recorre una lista con todas las parejas que se fueron creando durante la ejecución y se eliminan todas aquellas que se han roto (la mujer ha dejado al hombre por otro) – O(n2) - |
| Extender | O(nn) | En el peor caso prueba todas las combinaciones posibles de parejas. Eso equivale a recorrer completamente un árbol n-ario (que posee nn nodos). Cada vez que crea una pareja, revisa si es estable la asignación -O(n) -, por lo que quedaría O(nn+1) que es similar a O(nn) (a menos que n sea pequeño). |
| Registrar | O(1) |  |
| Olvidar | O(1) |  |
| ¿Listo? | O(n) | Pregunta si la asignación es estable -O(n)- y las parejas son n -O(1)-. |
| ¿Es estable? | O(n) en el peor caso,  O(cantActualdeParejas) en el caso general. | Revisa la estabilidad comparando la nueva pareja con las demás (que ya han sido verificadas entre sí), con lo que en el peor caso, hará n-1 comparaciones. |
| ¿Es Bloqueante(Pareja P1, Pareja P2)? | O(1) | Siempre hace la misma cantidad de comparaciones. |
| LOS DEMÄS MËTODOS | O(1) |  |

D – Respuestas/Conclusiones

**2) b)** Que la asignación es completa es muy simple de probar, ya que el algoritmo continúa haciendo rondas en tanto haya hombres solteros. Ergo, no va a terminar hasta que todos los hombres, y por ende las mujeres, estén casados.

Respecto a la estabilidad, ésta está garantizada por el modo en que arma la asignación. Como los hombres siempre se le declaran a la mujer que más prefieren que no los ha rechazado, y las mujeres siempre eligen la mejor propuesta que reciben, resulta que al final del algoritmo todos estarán con la mejor pareja, que no necesariamente será la número uno en su ranking, sino la mejor posible dada la configuración de preferencias. Por otra parte, no puede ocurrir que haya un par bloqueante: puede que haya un hombre que prefiera estar con otra mujer, pero no puede ocurrir que ella también lo prefiera a él. ¿Por qué? Porque si el hombre prefería a esa mujer sobre su compañera actual, en algún momento se le declaró y fue rechazado. Sin embargo, es imposible que ella lo haya rechazado para quedarse con otro que prefiere menos, porque las mujeres siempre eligen la mejor propuesta que reciben. Supóngase ahora que existe una mujer que preferiría estar con otro hombre. Sin embargo, ese hombre no puede estar con otra que le guste menos, porque si la primera mujer lo prefería a él hubiera aceptado su propuesta antes que la de otro que le gustara menos. En suma, como no puede haber ningún par bloqueante, la asignación final es estable.

**c)** El algoritmo termina cuando no hay hombres solteros. Razonando por el absurdo, supóngase que el algoritmo jamás termina. Ergo, lo que ocurre en ese caso es que al menos un hombre es rechazado indefinidamente. Pero eso es imposible, por lo siguiente: En el caso de que se hayan formado N-1 parejas, sólo quedarán solteros un hombre y una mujer. ¿Puede ocurrir que ella lo rechace? No, porque él será su único pretendiente (los demás no se le declararán porque ya están comprometidos); dado que la mujer siempre acepta la mejor propuesta, en este caso la aceptará, porque es la última.

¿Puede pasar que, aunque ella lo acepte, la asignación resulte inestable? En este punto, cabe considerar que al final de la k-ésima etapa, la asignación transitoria es estable (obviando a los solteros, claro), por las características del algoritmo: es fácil probarlo usando el mismo razonamiento del inciso anterior. Entonces, al llegar al punto donde hay N-1 parejas formadas, éstas son estables entre sí: los únicos que pueden originar conflicto son los dos solterones. En primer lugar, ¿Puede pasar que al hombre le guste más otra mujer que la que lo aceptó al final (porque no le quedaba otra)? Sí, pero ninguna de las mujeres casadas que le gusten más que su compañera preferirá estar con él a estar con su marido, por la sencilla razón de que cuando él se les declaró, lo rechazaron y se quedaron con un pretendiente que les gustaba más. En suma, no hay inestabilidad por parte del hombre. Ahora hay que considerar a la mujer. Llegada la situación final (N-1 parejas y 2 solteros), puede haber ocurrido lo siguiente:

\* Ella nunca recibió ninguna propuesta hasta ese momento: En ese caso, no hay hombres casados que la prefieran sobre sus parejas actuales, porque de no ser así se le habrían declarado. Entonces, aunque a ella le guste más otro hombre que el que se le propone al final, ningún otro querrá divorciarse para huir con ella, así que no habrá inestabilidad.

Como conclusión, cada vez que se llega a N-1 parejas formadas, en él próximo paso los dos solteros restantes se comprometerán y terminará el proceso, dando lugar a una asignación completa y estable (por el inciso b).

Por último, es menester probar que siempre se llega a dicha situación. En cada ronda, todos los hombres solteros se declararán y al menos uno será aceptado, porque lo peor que podría ocurrir es que todos se le declaren a la misma, quien se quedará con el mejor para ella. O sea que en cada ronda se forma al menos una pareja: el historial de rechazos evita que los hombres sigan insistiendo si son rechazados, así que no habrá situaciones cíclicas donde se romperá una pareja para formar otra y nunca se avanzará. Entonces, siempre se llegará a la instancia definitiva.

**d)** Por simétrico, se entiende que, dadas las mismas preferencias, los resultados son los mismos si en el algoritmo las mujeres y los hombres intercambian roles, y ello es cierto para todas las entradas posibles. La respuesta es no, y para demostrarlo alcanza con un contraejemplo, relativamente trivial. Aplicando Gale-Shapley a la siguiente configuración inicial...

| H/M | 1(favorita) | 2 | 3 |
| --- | --- | --- | --- |
| Juan | Juana | Pepa | Josefa |
| Pepe | Pepa | Josefa | Juana |
| José | Josefa | Juana | Pepa |

| M/H | 1(favorito) | 2 | 3 |
| --- | --- | --- | --- |
| Juana | Pepe | José | Juan |
| Pepa | José | Juan | Pepe |
| Josefa | Juan | Pepe | José |

...Se obtiene:

| Los hombres se declaran (GS) | | Las mujeres se declaran(GS con roles invertidos) | |
| --- | --- | --- | --- |
| Juan | Juana | Juana | Pepe |
| Pepe | Pepa | Juan | Pepe |
| José | Josefa | Pepe | José |
| **Grado Satisf. H = 9 (máximo)**  **Grado Satisf. M = 3 (mínimo)** | | **Grado Satisf. H = 3 (mínimo)**  **Grado Satisf. M =9 (máximo)** | |

En líneas generales (puede haber excepciones), la asignación siempre tenderá a favorecer a quienes se declaren primero, o sea los hombres en la versión usual de Gale-Shapley. Para obtener la solución que favorezca más a las mujeres, basta con intercambiar los roles: que las mujeres se declaren primero y los hombres vayan aceptando las mejores propuestas.

e) Asumiendo que lo único que una mujer puede hacer para alcanzar su objetivo es permutar su lista de preferencias, se demuestra que a veces es posible que fuerce el algoritmo (que es óptimo para los hombres) y logre obtener a su hombre favorito. Está claro que si no tuviera que entregar una lista completa, simplemente podría declarar a los hombres con ranking inferior al que desea como inaceptables, y no incluirlos en la lista. Por otra parte, ella conoce las preferencias de los demás individuos, las cuales permanecen fijas.

Sea P(m) = {h1, h2, ...hn} la verdadera lista de preferencias de m, la mujer tramposa, y sea P(h, m) la lista que ella entregará a la agencia a fin de obtener a su hombre deseado. Puede ocurrir que ello sea imposible, en cuyo caso P(h,m) será nula. Entonces, lo que la mujer puede hacer es:

1. Correr el algoritmo GS orientado a hombres, con P(m), “en su cabeza” y llevar cuenta de todos los hombres que se le declaran. Sea H el compañero que ella obtendría. Sea, por el momento, P(h,m) = P(h).
2. Si hj es el j-ésimo hombre que se le propuso, ella puede colocar a hj en el tope de su lista, obteniendo la lista P(hj,m) para que cuando se corra de nuevo GS y él se le declare, él resulte su compañero al final de la ejecución. Se dice entonces que hj es una pareja potencial de m.
3. Obtener las listas “falsas” para todos los otros hombres que se le propusieron durante esa primera “corrida”, salvo para H. Se dice entonces que se ha agotado al hombre H, que es el que obtendría con el algoritmo orientado a hombres.
4. Sea u cualquier otro hombre. Repetir lo hecho para m para u, identificando, todas las parejas potenciales para m en todos los casos. Sea N el conjunto de todas las parejas potenciales (y por lo tanto agotadas) de m.
5. Sea HO el hombre que más le gusta a m. Claramente, HO pertenece al conjunto N. Entonces, P(HO, m) es la lista de preferencias que la mujer debe entregar para quedarse con HO

Se puede demostrar la verdad de la última proposición, pero ello presenta bastantes dificultades teóricas.

f) Es un algoritmo voraz/”greedy”, porque procede por etapas sucesivas donde se elige lo mejor en ese momento (los hombres solteros se le declaran a sus preferidas y las mujeres aceptan la mejor propuesta), y la solución final, aunque siempre es estable, no necesariamente es la mejor posible (nuevamente, ello puede verse con un contraejemplo donde se conozca la mejor solución). Vale aclarar que la calidad de una solución se puede medir, por ejemplo, con el promedio entre el grado de satisfacción de los hombres y el de las mujeres. A su vez, el grado de satisfacción se obtiene sumando para todos los individuos de un sexo N-el ranking de su pareja (ya que este ranking va de 1-el/la máximo/a favorito/a- a N).

CONCLUSIONES

En definitiva, el algoritmo de Gale y Shapley no siempre ofrece la mejor solución, pero obtiene una en un tiempo mucho mejor que el de Backtracking. Si se quisiera la mejor solución posible, habría que usar backtracking para obtener todas las posibles, y seleccionar la mejor entre todas ellas. Está claro que eso sería un poco más costoso que O(nn), en tanto Gale-Shapley es O(n2). Resulta entonces que para n un poco grande (mayor o igual que 10), obtener la mejor solución “maximizando Bactracking” es casi impracticable. Tal vez se podría modificar Gale – Shapley para que devuelva la mejor solución, pero ello le haría perder su carácter ávido y es casi seguro que el orden empeoraría. Elegir la forma de resolver el problema dependerá de cuánto tiempo se esté dispuesto a sacrificar.

**E- CÓDIGO FUENTE**

----------------------------------------------------------------------

CLASE AGENCIA

import java.util.ArrayList;

import java.util.Iterator;

/\*

\* Esta clase representa a una agencia que contiene la información de todos

\* los hombres y mujeres (incluido su ranking).

\* Además, la agencia es la que se encarga de asignarle un número único a los

\* hombres o a las mujeres.

\*/

public class Agencia{

private ArrayList hombres;

private ArrayList mujeres;

private ArrayList parejas; //Solución transitoria

private ArrayList solucion; //Solución definitiva.

private int cantParejas; //Cuenta la cantidad de parejas no rotas de la colección "parejas"

//(usado en el algoritmo de Gale & Shapley).

public Agencia (){

hombres = new ArrayList();

mujeres = new ArrayList();

parejas = new ArrayList();

cantParejas = 0;

}

public Individuo getHombre(int ID){

return (Individuo) hombres.get(ID);

}

public Individuo getMujer(int ID){

return (Individuo) mujeres.get(ID);

}

public int cantHombres(){

return hombres.size();

}

public int cantMujeres(){

return mujeres.size();

}

public void addHombre(Individuo hombre){

hombre.setNumero(hombres.size());

hombres.add(hombre);

}

public void addMujer(Individuo mujer){

mujer.setNumero(mujeres.size());

mujeres.add(mujer);

}

/\*

\*----------------------------------------

\* RESOLVIENDO POR GALE & SHAPLEY

\*----------------------------------------

\*/

/\*

\* PRE: todos los hombres deben tener un ranking de mu-

\* jeres y viceversa.

\*/

public ArrayList asignarConGS (){

parejas.clear();

//PRIMERA ETAPA

//Cada hombre elige a su mujer favorita.

for(int i=0;i<this.cantHombres();i++){

Individuo hombre = this.getHombre(i);

hombre.mejorPretendido().addPretendiente(hombre);

//El hombre es ahora pretendiente de su mujer favorita.

}

//Cada mujer se empareja con el pretendiente que más le agrade.

for(int i=0; i<this.cantMujeres();i++){

Individuo mujer = this.getMujer(i);

Individuo hombre = mujer.pretendienteFavorito();

if (hombre != null) { //Si alguien se le había declarado a la mujer.

Pareja pareja = new Pareja(hombre,mujer);

parejas.add(pareja);

cantParejas++;

}

mujer.olvidarPretendientes();

}

//RESTO DE LAS ETAPAS

while(!listoGS()){//Mientras no esté lista la asignación...

//Cada hombre elige a la mujer favorita que no lo ha rechazado aún.

for(int i=0;i<this.cantHombres();i++){

Individuo hombre = this.getHombre(i);

if(hombre.isSoltero()){

Individuo mujer = hombre.mejorPretendido();

/\*El hombre es ahora pretendiente de la mujer preferida que no lo

rechazó aún.\*/

mujer.addPretendiente(hombre);

if (!mujer.isSoltero()) mujer.addPretendiente(mujer.getPareja().getHombre());

/\*La última línea es para que cuando mujer devuelva su preten-

diente favorito, también tenga en cuenta a su marido.\*/

}

}

//Cada mujer elige entre los pretendientes y su marido (si tiene).

for(int i=0; i<this.cantMujeres();i++){

Individuo mujer = this.getMujer(i);

Individuo hombre = mujer.pretendienteFavorito();

if (hombre != null) {

//Si se le declaró alguien en esta ronda a la mujer.

//Si la mujer tiene pareja, que rompa con ella.

if (!mujer.isSoltero()){

mujer.getPareja().romper();

cantParejas--;

}

Pareja pareja = new Pareja(hombre,mujer);

parejas.add(pareja);

cantParejas++;

mujer.olvidarPretendientes();

}

}

}

quitarParejasRotas (parejas);

/\*El último paso se debe a que "parejas" está llena de parejas que han ro-

to y no fueron removidas de allí. Esto se debe a que es más eficiente

hacer sólo un recorrido a la colección removiendo todas las parejas rotas

que ir haciéndolos durante la ejecución del algoritmo.\*/

solucion = parejas;

return solucion;

}

/\*

\* La asignación está lista solo si se han emparejado todos los hombres (y,

\* por ende, todas las mujeres).

\*/

private boolean listoGS(){

return(cantParejas == this.cantHombres());

}

/\*

\* Quito de "parejas" todas las parejas que se hayan separado (las identifico

\* porque la mujer o el hombre son "null".

\*/

private void quitarParejasRotas(ArrayList parejas){

Iterator it = parejas.iterator();

while (it.hasNext()){

Pareja p = (Pareja) it.next();

if ((p.getHombre()==null)||(p.getMujer()==null)) it.remove();

}

}

/\*

\*----------------------------------------

\* RESOLVIENDO POR BACKTRACKING

\*----------------------------------------

\*/

/\*

\* PRE: todos los hombres deben tener un ranking de mu-

\* jeres y viceversa.

\*/

public ArrayList asignarConBT (){

parejas.clear();

solucion = null;

extender(0);

return solucion;

}

/\*

\* Parte principal de la solución por backtracking.

\*/

private void extender (int hombreActual){

if (!this.listoBT()){

for (int mujer = 0;mujer < this.cantMujeres();mujer++){ //Pruebo todas las posibilidades

Pareja parejaNueva = new Pareja (this.getHombre(hombreActual),this.getMujer(mujer));

registrar(parejaNueva);

if (esEstable())

extender(hombreActual+1);

if (solucion != null) return;

olvidar();

}

}else solucion = parejas;

}

private void registrar(Pareja p){

parejas.add(p);

}

private void olvidar(){

parejas.remove(parejas.size() - 1);

}

/\*

\* La asignación está lista si es factible y además todos los hombres (y, por

\* ende, las mujeres) están en pareja.

\*/

private boolean listoBT(){

return(esEstable() && (parejas.size() == this.cantHombres()));

}

/\*

\* Devuelve true si la asignación es estable (no tiene pares bloqueantes).

\*/

private boolean esEstable(){

if (parejas.size() ==0) return true; //Una asignación vacía es estable.

/\* Dado que las n-1 parejas ya han sido chequeadas entre sí, sólo hay que

\* verificar la estabilidad entre la nueva pareja y todas las demás. \*/

Pareja p = (Pareja) parejas.get(parejas.size()-1); //La última pareja agregada

for(int i=0; (i+1)<parejas.size() ;i++){

if (this.esBloqueante((Pareja) parejas.get(i),p)) return false;

}

return true;

}

/\*

\* Devuelve "true" si y solo si un hombre y una mujer de las dos parejas

\* distintas preferiría estar casado entre sí en lugar de con su compañero

\* actual.

\*/

private boolean esBloqueante(Pareja p1, Pareja p2){

Individuo h1 = p1.getHombre();

Individuo m1 = p1.getMujer();

Individuo h2 = p2.getHombre();

Individuo m2 = p2.getMujer();

if ((h1 == h2) || (m1 == m2)) return true;

/\* Si al hombre 1 le gusta más la mujer 2 que su esposa, la mujer 1, y

\* a la mujer 2 le ocurre lo mismo con él, hay inestabilidad.\*/

if ((h1.getPuesto(m2) < h1.getPuesto(m1)) &&

(m2.getPuesto(h1) < m2.getPuesto(m2)))

return true;

/\* Si a la mujer 1 le gusta más el hombre 2 que su marido, el hombre 1, y

\* al hombre 2 le ocurre lo mismo con ella, hay inestabilidad.\*/

if ((m1.getPuesto(h2) < m1.getPuesto(h1)) &&

(h2.getPuesto(m1) < h2.getPuesto(m2)))

return true;

/\* En caso contrario, la asignacion consistente en sólo estas dos parejas

\* es estable\*/

return false;

}

}

----------------------------------------------------------------------

CLASE ARCHIVOENTRADA

import java.io.\*;

import java.lang.Character;

public class ArchivoEntrada {

private FileReader lector;

private BufferedReader entrada;

private String nomArch;

//Los siguientes atributos representan el caracter devuelto cuando hay un fin

//de archivo o algun error.

public final static String EOFstr = "" + Character.MAX\_VALUE;

public final static char EOFchr = Character.MAX\_VALUE;

/\*

\* Averiguar cuales son la cadena vacía que se devuelve al leer una línea en blanco

\* o el caracter de fin de línea (o el 1º y el último de una línea en blanco).

\* OBS: creo que son:

\* TEXTO\n

\* \r\n

\* \rTEXTO\n

\* \rTEXTO...

\*/

public ArchivoEntrada(String nomArch) throws IOException{

this.nomArch = nomArch;

lector = new FileReader (nomArch);

entrada = new BufferedReader(lector);

}

/\*

\* Lee una línea del archivo de entrada.

\* Si ocurre un error, devuelve una cadena vacía.

\*/

public String readln(){

try{

return entrada.readLine();

}catch (IOException e){

return EOFstr;

}

}

/\*

\* Lee un caracter del archivo de entrada.

\* Si ocurre un error, devuelve el caracter nulo.

\*/

public int read(){

try{

return entrada.read();

}catch(IOException e){

return EOFchr;

}

}

/\*

\* Se ubica en el primer caracter del archivo.

\*/

public void reset(){

try{

//Cierro el archivo y lo vuelvo a abrir.

entrada.close();

lector = new FileReader (nomArch);

entrada = new BufferedReader(lector);

}catch(IOException e){

e.printStackTrace(System.err);

}

}

}

----------------------------------------------------------------------

CLASE ARCHIVOSALIDA

import java.io.BufferedWriter;

import java.io.FileNotFoundException;

import java.io.FileWriter;

import java.io.IOException;

import java.io.PrintWriter;

public class ArchivoSalida {

/\*

\*------------------------------------------------------------

\*Constructores:

\*------------------------------------------------------------

\*/

public ArchivoSalida(String nomArch){

this.nomArch = new String (nomArch);

archivo = abrir (nomArch,false);

}

public ArchivoSalida(String nomArch, boolean append){

this.nomArch = new String (nomArch);

archivo = abrir (nomArch,append);

}

/\*

\*Guarda los cambios realizados hasta el momento en

\*el archivo.

\*/

public void guardarCambios(){

archivo.close();

archivo = abrir (nomArch,true);

}

/\*

\*Imprime una línea en el archivo y genera un

\*caracter "fin de línea".

\*/

public void println(boolean b){archivo.println(b);}

public void println(char c) {archivo.println(c);}

public void println(char[] s) {archivo.println(s);}

public void println(double d) {archivo.println(d);}

public void println(float f) {archivo.println(f);}

public void println(int i) {archivo.println(i);}

public void println(long l) {archivo.println(l);}

public void println(Object o) {archivo.println(o.toString());}

/\*

\*Imprime una cadena de caracteres en el archivo y no

\*genera el "fin de línea".

\*/

public void print(boolean b) {archivo.print(b);}

public void print(char c) {archivo.print(c);}

public void print(char[] s) {archivo.print(s);}

public void print(double d) {archivo.print(d);}

public void print(float f) {archivo.print(f);}

public void print(int i) {archivo.print(i);}

public void print(long l) {archivo.print(l);}

public void print(Object o) {archivo.print(o.toString());}

/\*

\*Abre el archivo nomArch preparándolo para la escritura.

\*Si "append == false", borra todo el contenido previo del

\*archivo que se abre.

\*/

private static PrintWriter abrir (String nomArch, boolean append){

try{

return new PrintWriter(new BufferedWriter(new FileWriter(nomArch,append)));

}

catch(FileNotFoundException archNoEnc){

System.out.println("Archivo no encontrado: " + archNoEnc);

}

catch(IOException ioExc){

ioExc.printStackTrace(System.err);

}

return null;

}

/\*

\*------------------------------------------------------------

\*Atributos:

\*------------------------------------------------------------

\*/

private PrintWriter archivo;

private String nomArch;

}

----------------------------------------------------------------------

CLASE INDIVIDUO

import parva.Parva;

/\*

\* Esta clase modela a un individuo conteniendo toda la información requerida

\* para resolver el problema de las asignaciones estables.

\*/

public class Individuo {

private String nombre;

private boolean hombre; //Indica si es o no un hombre.

private int ID;

private Pareja pareja; //Pareja a la cual pertenece éste individuo.

private int[] posiciones; //Posiciones en el ranking.

private int cantRanqueados;

private Parva pretendidos; //ID de los individuos pretendidos ordenados por puesto.

private Parva pretendientes; //Individuos que se le declararon a éste.

public Individuo (String nombre, boolean hombre){

this.nombre = nombre;

this.hombre = hombre;

pareja = null;

//Cantidad estimativa de la cantidad de individuos: 1000.

cantRanqueados = 1000;

posiciones = new int[cantRanqueados];

pretendientes = new Parva(1000, new Integer(Integer.MIN\_VALUE));

pretendidos = new Parva(1000, new Integer(Integer.MIN\_VALUE));

}

public Individuo (String nombre, boolean hombre,int totIndividuos){

this.nombre = nombre;

this.hombre = hombre;

pareja = null;

//Cantidad estimativa de la cantidad de individuos: 1000.

cantRanqueados = totIndividuos;

posiciones = new int[cantRanqueados];

pretendientes = new Parva(totIndividuos, new Integer(Integer.MAX\_VALUE));

pretendidos = new Parva(totIndividuos, new Integer(Integer.MAX\_VALUE));

}

public String getNombre(){

return nombre;

}

public int getNumero(){

return ID;

}

public Pareja getPareja(){

return pareja;

}

public boolean isHombre(){

return hombre;

}

public boolean isSoltero(){

return (pareja == null);

}

/\*

\* Devuelve el puesto dado por el individuo a i.

\* Y, en caso de haber un error, por no haber asignado un puesto a

\* i, se devuelve -1.

\*/

public int getPuesto(Individuo i){

if (i.getNumero() < 0) return -1;

if (i.getNumero() > cantRanqueados) return -1;

return posiciones[i.getNumero()];

}

/\*

\* Devuelve el individuo preferido por éste. En la próxima llamada, devolverá el

\* que lo sigue.

\*/

public Individuo mejorPretendido(){

return (Individuo) pretendidos.getMaximo();

}

/\*

\* Devuelve el pretendiente con mejor puesto para este individuo.

\*/

public Individuo pretendienteFavorito(){

return (Individuo)pretendientes.primero();

}

public void olvidarPretendientes(){

pretendientes.borrar();

}

public void setNombre(String nombre){

this.nombre = nombre;

}

public void setNumero(int ID){

this.ID = ID;

}

public void formarPareja(Pareja pareja){

this.pareja = pareja;

}

public void separar(){

pareja = null;

}

public void setHombre(boolean hombre){

this.hombre = hombre;

}

/\*

\* Si dos individuos tienen el mismo número, no serán distinguidos por

\* este método.

\* Si se pasa por parámetro un puesto negativo, será asignado como

\* cero.

\*/

public void setPuesto(Individuo i, int puesto){

if (i.getNumero() > cantRanqueados){

//Aumento el tamaño del vector para que tenga más cantidad de posiciones

//de tal modo que exista una posición posiciones[i.getNumero()]

//pero, de paso, aprovecho y lo hago más grande aún para prevenir

//futuros aumentos de tamaño ya que son costosos.

int nuevoTam= Math.round(i.getNumero() \* 1.5f);

int[] nuevo = new int[nuevoTam];

for (int j=0; j<cantRanqueados; j++) nuevo[j] = posiciones[j];

cantRanqueados = nuevoTam;

posiciones = nuevo;

}

posiciones[i.getNumero()] = Math.max (puesto,0);

pretendidos.insertar(i,new Integer(puesto));

}

public void addPretendiente(Individuo i){

pretendientes.insertar(i,new Integer(this.getPuesto(i)));

}

}

----------------------------------------------------------------------

CLASE PAREJA

/\*

\* Clase que representa una pareja entre un hombre y una mujer.

\*/

public class Pareja{

private Individuo hombre;

private Individuo mujer;

public Pareja (Individuo hombre,Individuo mujer){

this.hombre = hombre;

this.mujer = mujer;

hombre.formarPareja(this);

mujer.formarPareja(this);

}

public Individuo getHombre(){

return hombre;

}

public Individuo getMujer(){

return mujer;

}

public void romper(){

hombre.separar();

mujer.separar();

hombre = null;

mujer = null;

}

}

----------------------------------------------------------------------

CLASE TABLAHASH

public class TablaHash {

private Individuo[] individuos;

private int tamanio;

private int cant; //Representa la cantidad de elementos de la tabla.

public TablaHash(){

individuos = new Individuo[1000];

tamanio = 1000;

cant = 0;

}

public TablaHash(int tamanio){

individuos = new Individuo[tamanio];

this.tamanio = tamanio;

cant = 0;

}

/\*

\* Devuelve el individuo cuyo nombre es el dado.

\* En caso de que no pertenezca a la tabla, devuelve null.

\* Para hallarlo se aplica el siguiente método:

\* 1\_ Calculo: "a = hashCode (nombre del individuo)".

\* 2\_ Si la ubicación está vacía, devuelvo null.

\* 3\_ Si la ubicación está ocupada por el individuo pedido, lo devuelvo.

\* 4\_ En otro caso, sumo uno a "a" y vuelvo al paso 2.

\*/

public Individuo get(String nombre){

int cod = hashCode (nombre);

do{

if (individuos[cod] == null) return null;

if (individuos[cod].getNombre().equals(nombre)) return individuos[cod];

cod++;

}while (true);

}

/\*

\* Agrega el individuo a la tabla.

\* Para elegir la ubicación en la que se coloca se siguen los siguientes pa-

\* sos:

\* 1\_ Calculo: "a = hashCode (nombre del individuo)".

\* 2\_ Si la ubicación está vacía, lo agrego.

\* 3\_ En otro caso, sumo uno a "a" y vuelvo al paso 2.

\* Si llevo la tabla a un nivel de ocupación de más del 70%, duplico su tama-

\* ño y reubico todos los elementos.

\*/

public void add(Individuo i){

int cod = hashCode (i.getNombre());

while (individuos[cod] != null){

cod++;

cod%= tamanio;

}

individuos[cod] = i;

cant++;

if (cant >= Math.round(tamanio \* 0.7f)) reHashear();

}

/\*

\*EN EL PROGRAMA QUE IMPLEMENTAMOS NO ES NECESARIO REMOVER ELEMENTOS.

public void remove(Individuo i){

return;

}

\*/

/\*

\* Devuelve el código de hash de un individuo dado su nombre.

\* codigo = c0 \* B^n + c1 \* B^(n-1) + ... + cn

\* Donde ci es el i-ésimo caracter del nombre (convertido a minúsculas y qui-

\* tándole los espacios que haya antes de la primera inicial o después de la

\* última letra).

\* B = 26 (de este modo, si no hubiera restricciones como el tamaño de la ta-

\* bla de hash o de un máximo representable por "int", la asignación del có-

\* digo sería biunívoca).

\*/

private int hashCode(String nombre){

//El nombre en minúsculas y sin espacios al comienzo o al final.

String cad = nombre.trim().toLowerCase();

int codigo = (int)(nombre.charAt(0));

for (int i=1;i<cad.length();i++){

codigo\*= 5;

codigo+= (nombre.charAt(0) - 'a') + 1;

codigo%= tamanio;

}

return codigo;

}

/\*

\* Duplica el tamanio de la tabla de hash.

\* Para logarlo, crea una nueva tabla del doble de tamanio que la actual y

\* copia todo el contenido de la vieja en ella.

\*/

private void reHashear(){

int cantCopiada = 0;

TablaHash aux = new TablaHash(tamanio \* 2);

for (int i = 0;(i < tamanio)||(cantCopiada < cant); i++)

if (individuos[i] != null){

aux.add(individuos[i]);

cantCopiada++;

}

this.individuos = aux.individuos;

}

}

----------------------------------------------------------------------

CLASE TDATP2

import java.util.ArrayList;

import java.io.IOException;

public class tdatp2{

/\*

\* Lectura del archivo con los datos de entrada.

\* El formato de la entrada es:

\* Hombre1: Mujer11,Mujer12,Mujer13,...,Mujer1N

\* ...

\* HombreN: MujerN1,MujerN2,MujerN3,...,MujerNN

\*

\* Mujer1: Hombre11,Hombre12,Hombre13,...,Hombre1N

\* ...

\* MujerN: HombreN1,HombreN2,HombreN3,...,HombreNN

\*

\* Donde N es la cantidad de hombres (y mujeres) y, MujerIJ es la mujer cali-

\* ficada con J por el hombre I (idem para HombreIJ).

\* Para realizar el parseo, tengo en cuenta el hecho de que con la primer lí-

\* nea del archivo puedo obtener los nombres de todas las mujeres (y, por lo

\* tanto, crearlas).

\* Luego, comienzo la lectura línea a línea agregando a la lista de preferen-

\* cias de cada hombre la mujer que aparezca (y en cada línea que leo debo

\* crear un hombre).

\* Cuando llega el momento de leer las preferencias de cada mujer ya tengo a

\* todos los hombres creados (los creé mientras leía sus preferencias).

\* Para tener acceso rápido al objeto "Individuo" que representa a cada hom-

\* bre o a cada mujer mientras leo el archivo (esto es, identificándolos por

\* su nombre) utilizo una tabla de hash.

\*/

public static Agencia parsear(String nomArch){

Agencia agencia = new Agencia();

ArchivoEntrada entrada = null;

try{

entrada = new ArchivoEntrada(nomArch);

}catch(IOException e){

System.out.println("ERROR EN LA ENTRADA.");

System.exit(1);//Sale del programa si no se puede parsear.

}

String primLineaH = entrada.readln(); //Primera linea con calificaciones de hombres

TablaHash mujeres = new TablaHash();

TablaHash hombres = new TablaHash();

//CREO TODAS LAS MUJERES:

int i = 0;

//Me salteo el nombre del primer hombre.

while (primLineaH.charAt(i)!=':') i++;

i+=2;//Me ubico en la inicial del nombre de la primera mujer.

//Obtengo el nombre de todas las mujeres.

int iAnt = i;

while (i <= primLineaH.length()){

if((i == primLineaH.length())||(primLineaH.charAt(i)==',')){

//Acabo de leer el nombre de una mujer => la agrego a la lista.

String nombreM = primLineaH.substring(iAnt,i);

Individuo mujer = new Individuo(nombreM,false);

mujeres.add(mujer);

agencia.addMujer(mujer);

iAnt = i+1;

}

i++;

}

//CARGO LOS HOMBRES CON SUS LISTAS DE PREFERENCIAS

entrada.reset();

String linea = entrada.readln();

while (!linea.equals("")){//Mientras no sea una línea vacía...

int calif = 0;

i = 0;

//Obtengo el nombre del hombre para crearlo.

while (linea.charAt(i)!=':') i++;

Individuo hombre = new Individuo (linea.substring(0,i),true);

i+=2;//Me ubico en la inicial del nombre de la primera mujer.

//Obtengo el nombre de cada mujer y la agrego a la lista de preferencias.

iAnt = i;

while (i<=linea.length()){

if((i == linea.length())||(linea.charAt(i)==',')){

//Acabo de leer el nombre de una mujer => la agrego a la lista.

String nombreM = linea.substring(iAnt,i);

Individuo mujer = mujeres.get(nombreM);

hombre.setPuesto(mujer,calif);

iAnt = i+1;

calif++;

}

i++;

}

hombres.add(hombre);

//La última línea es para que cuando cargue a las mujeres ya tenga a

//todos los hombres cargados.

agencia.addHombre(hombre);

linea = entrada.readln();

}

//CARGO LAS MUJERES CON SUS LISTAS DE PREFERENCIAS

//Valen los mismos comentarios que para la carga de los hombres.

linea = entrada.readln();

while (linea != null){//Mientras no llegue al fin de archivo...

int calif = 0;

i = 0;

while (linea.charAt(i)!=':') i++; //Obtengo el nombre de la mujer.

Individuo mujer = mujeres.get(linea.substring(0,i));

i+=2;

iAnt = i;

while (i<=linea.length()){

if((i == linea.length())||(linea.charAt(i)==',')){

String nombreH = linea.substring(iAnt,i);

Individuo hombre = hombres.get(nombreH);

mujer.setPuesto(hombre,calif);

iAnt = i+1;

calif++;

}

i++;

}

linea = entrada.readln();

}

return agencia;

}

/\*

\* Escritura del archivo con las soluciones.

\* La salida es del modo:

\* ------------------------------

\* Cantidad de parejas: XXXX

\* Tiempo requerido: XXXXms

\* -----------SOLUCIÓN-----------

\* #1: Hombre1 - Mujer1

\* #2: Hombre2 - Mujer2

\* ...

\* #n: Hombren - Mujern

\*/

public static void salida(String nomArch, ArrayList parejas, long tiempo){

ArchivoSalida salida = new ArchivoSalida (nomArch);

String linea = "------------------------------";

salida.println(linea);

System.out.println(linea);

linea = "Cantidad de parejas: " + parejas.size();

salida.println(linea);

System.out.println(linea);

linea = "Tiempo requerido: " + tiempo + "ms";

salida.println(linea);

System.out.println(linea);

linea = "-----------SOLUCIÓN-----------";

salida.println(linea);

System.out.println(linea);

for (int i=0; i<parejas.size(); i++){

Pareja p = (Pareja)parejas.get(i);

linea = "#" + (i+1) + ": ";

linea+= p.getHombre().getNombre();

linea+= " - ";

linea+= p.getMujer().getNombre();

salida.println(linea);

System.out.println(linea);

}

salida.guardarCambios();

}

public static void main (String[] args){

int RANGO = 6;

long Ti,Tf;

ArrayList parejas = new ArrayList();

Agencia agencia = parsear("pruebas/prueba"+RANGO+".txt");

//Se llama al recolector de basura antes de comenzar, por las dudas.

System.gc();

//Comienza la medición del tiempo (se desprecia el parseo).

Ti = System.currentTimeMillis();

parejas = agencia.asignarConBT();

Tf = System.currentTimeMillis(); // Finaliza la medición del tiempo.

salida("resultados/sc\_BT"+RANGO+".txt",parejas,Tf-Ti);

//Rompo todas las parejas para que los individuos estén solteros al comenzar el

//algoritmo de Gale & Shapley.

for (int i = 0;i<parejas.size();i++){

Pareja p = (Pareja) parejas.get(i);

p.romper();

}

System.gc();

Ti = System.currentTimeMillis();

parejas = agencia.asignarConGS();

Tf = System.currentTimeMillis();

salida("resultados/sc\_GS"+RANGO+".txt",parejas,Tf-Ti);

}

}

----------------------------------------------------------------------

CLASE ELEMENTO

package parva;

public class Elemento {

private Object dato;

private Comparable prioridad;

public Elemento(Object d, Comparable p){

this.dato = d;

this.prioridad = p;

}

public Object getDato(){

return dato;

}

public Comparable getPrioridad(){

return prioridad;

}

public void setDato(Object d){

dato = d;

}

public void setPrioridad(Comparable p){

prioridad = p;

}

}

----------------------------------------------------------------------

CLASE PARVA

package parva;

/\*\*

\* Parva/montículo/heap de mínimos. Requiere conocer, para la

\* creación, la cantidad mínima de elementos. De todos modos, la

\* estructura puede crecer dinámicamente.

\*/

public class Parva {

private Elemento[] vector;

private int tamAct; //Cantidad actual de elementos.

public Parva(int n, Comparable infinito){

vector = new Elemento[n+1];

tamAct = 0;

vector[0] = new Elemento(null, infinito);

/\* vector[0] es un padre falso de la raíz, una cota superior de

\* todas las prioridades. Su fin es la simplificación de ciertas

\* operaciones.

\*/

for(int i=1;i<=n;i++){

vector[i] = null;

}

}

public boolean esVacia(){

return (tamAct==0);

}

public void borrar(){

for (int i=1;i<tamAct;i++)

vector[i] = null;

tamAct = 0;

}

/\*\*

\* PRE: Parva creada.

\* @return Los datos asociados al elemento de mayor prioridad.

\* Si la parva está vacía, devuelve "null".

\*/

public Object primero(){

if (vector[1] == null) return null;

return vector[1].getDato();

}

public void insertar(Object o, Comparable prior){

if (tamAct == vector.length-1)

// La estructura puede crecer en tiempo de ejecución.

duplicarVector();

int hueco = tamAct+1;

/\* El elemento sube en el "árbol" en tanto su prioridad sea menor que la

\* de su "padre actual". \*/

while(((vector[hueco/2].getPrioridad()).compareTo(prior))>0){

vector[hueco] = vector[hueco/2];

hueco/=2;

}

vector[hueco] = new Elemento(o, prior);

tamAct++;

}

/\*\*

\* PRE: Parva creada y vacía.

\* Aumenta al doble el espacio para insertar elementos dentro

\* de la parva.

\*/

private void duplicarVector(){

Elemento[] nuevoVector;

nuevoVector = new Elemento[vector.length\*2];

for(int i=1;i<vector.length;i++){

nuevoVector[i] = vector[i];

}

vector = nuevoVector;

}

/\*\*

\* PRE: Parva creada.

\*/

public Object getMaximo(){

if (vector[1] == null) return null;

Elemento e = (Elemento)vector[1];

vector[1] = vector[tamAct];

// El último elemento es ahora la raíz.

vector[tamAct] = null;

tamAct--;

/\* Ahora hay que restablecer la propiedad heap, de ser

\* ello necesario. \*/

hundir(1);

return e.getDato();

}

/\*\*

\* Es el swapdown. "Hunde" al elemento en la parva, manteniendo

\* la propiedad heap.

\*/

private void hundir(int hueco){

int hijo = hueco\*2;

if (hijo > tamAct) return;

if (hijo+1<=tamAct)

if (vector[hijo].getPrioridad().compareTo(vector[hijo+1].getPrioridad())<0)

hijo++;

if (vector[hijo].getPrioridad().compareTo(vector[hueco].getPrioridad())<0) {

Elemento aux = vector[hijo];

vector[hijo] = vector[hueco];

vector[hueco] = aux;

hundir(hijo);

}

}

/\*\*

\* La inserción es ordenada, por lo que este método no sería necesario.

\* Sólo está por cuestiones de extensibilidad.

\*/

private void arreglarHeap(){

for(int i=tamAct/2;i>0;i--){

hundir(i);

}

}

}