UNIVERSITAT JAUME·I

1º Ingeniería Informática
1º Matemática Computacional

EI1008/MT1008 Programación II

Práctica 4: Algoritmos y complejidad computacional

1. Ejercicios sobre recursión

Ejercicio 1

Vamos a implementar un método estático que escriba todas las variaciones con repetición de los enteros de 1 a n tomados de m en m. Las variaciones con repetición de n elementos tomados de m en m son todas las posibles secuencias de longitud m que se pueden formar con n elementos. Por ejemplo, con los dígitos del uno al cuatro podemos hacer las siguientes variaciones con repetición tomándolos de tres en tres:

{111, 112, 113, 114, 121, 122, 123, 124, 131, 132, 133, 134, 141, 142, 143, 144, 211, 212, 213, 214, 221, 222, 223, 224, 231, 232, 233, 234, 241, 242, 243, 244, 311, 312, 313, 314, 321, 322, 323, 324, 331, 332, 333, 334, 341, 342, 343, 344, 411, 412, 413, 414, 421, 422, 423, 424, 431, 432, 433, 434, 441, 442, 443, 444}

Nuestro método tendrá el perfil:

```
public static void variacionesRepetición(int n, int m)
```

Si te fijas en el ejemplo, podemos formar las variaciones tomando los elementos de tres en tres si añadimos cada uno de los dígitos posibles al principio de las variaciones tomando los elementos de dos en dos. Esto nos sugiere una implementación recursiva. Si sabemos qué prefijo tenemos, podemos hacer lo siguiente:

- Si no hay que tomar elementos (`m == 0`), escribimos el prefijo.
- Si hay que tomar elementos, repetimos por cada elemento una ejecución recursiva añadiendo ese elemento al prefijo.

Como tenemos el problema del prefijo, vamos a utilizar un método auxiliar recursivo con perfil

```
private static void variacionesRepetición(int n, int m, String prefijo)
```

De esta manera, el cuerpo de nuestro método inicial es simplemente una llamada al auxiliar con el prefijo vacío:

```
public static void variacionesRepetición(int n, int m) {
    variacionesRepetición(n, m, "");
}
```

Y el del método recursivo será:

```
if (m == 0) // Caso base: escribir el prefijo
  ...
else // Caso general:
    for (int i = 1 ; i <= n ; i++)
        variacionesRepetición(n, m - 1, prefijo + i);
```

Termina de escribir las funciones y prepara un método `main` para comprobar que funcionan correctamente.

Ejercicio 2

En este ejercicio vamos a hacer algo parecido al anterior, pero en este caso estamos interesados en las permutaciones de n elementos. Como sabes, estas son las posibles maneras de ordenar los n elementos sin repetirlos y usándolos todos. Por ejemplo, las permutaciones de los números del uno al cuatro son:

{1234, 1243, 1324, 1342, 1423, 1432, 2134, 2143, 2314, 2341, 2413, 2431, 3124, 3142, 3214, 3241, 3412, 3421, 4123, 4132, 4213, 4231, 4312, 4321}

Nuestro método tendrá el perfil:

```
public static void permutaciones(int n)
```

Observando el ejemplo, puedes darte cuenta de que podemos seguir una estrategia similar al ejercicio anterior. Llevamos un prefijo acumulado y elegimos un elemento para añadir al final. Sin embargo, hay una diferencia importante: supongamos que el prefijo es `23`; entonces, las permutaciones que generemos recursivamente para completar el prefijo no podrán tener ni el dígito `2` ni el `3`. Esto nos obliga a que el método auxiliar tenga como perfil el siguiente:

```
private static void permutaciones(int[] disponibles, String prefijo)
```

El cuerpo del método inicial preparará el vector `disponibles` con todos los números de `1` a `n` y hará la llamada al auxiliar:

```
public static void permutaciones(int n) {
    int[] disponibles = new int[n];
    ... // Rellenar disponibles
    permutaciones(disponibles, "");
}
```

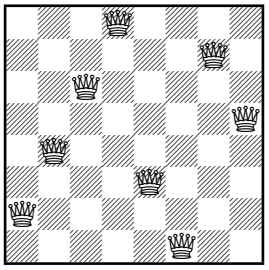
En cuanto al método auxiliar, el caso base ocurrirá cuando la longitud de `disponibles` sea 0, momento en que escribiremos el prefijo. En el caso general, tendremos que coger cada uno de los elementos de `disponibles`, rellenar un segundo vector con el resto de elementos y hacer la llamada recursiva:

```
if (disponibles.length == 0) // Caso base: escribir el prefijo
  ...
else { // Caso general
       int[] auxiliar = ...;
       for (int e: disponibles) {
           // Rellenar auxiliar con los disponibles distintos de e
           ...
           permutaciones(auxiliar, prefijo + e);
       }
}
```

Termina de escribir las funciones y prepara un método `main` para comprobar que funcionan correctamente.

Ejercicio 3

Vamos a resolver el problema de colocar ocho reinas en un tablero de ajedrez sin que se ataquen entre sí. Este es un problema clásico del que puedes encontrar numerosas referencias[1]. Un ejemplo de solución es la siguiente posición:



Simbólicamente, podemos escribirla dando por cada columna del tablero el número de fila ocupada por la reina correspondiente. Esa solución sería la `24683175`. Es fácil darse cuenta de que cada solución tiene que ser una permutación de los números del uno al ocho (si hubiera algún número repetido, habría dos reinas atacándose). Pero no basta con que sea una permutación. Es necesario, además, evitar ataques en diagonal. Si dos reinas están en las posiciones (i_1, j_1) e (i_2, j_2), se atacarán en diagonal si $|i_1 - i_2| = |j_1 - j_2|$.

Prepara una copia de tu solución al problema del ejercicio anterior. En este caso, el método principal será:

```
public static void ochoReinas() {
    int[] filas = new int[8];
    ... // Rellenar con los números del uno al 8
    ochoReinas(filas, "");
}
```

También tendrás que hacer un método que compruebe si la solución es correcta:

```
private static boolean esCorrecta(String solución)
```

Ahora, modifica el método auxiliar para que sea:

```
public static void ochoReinas(int[] disponibles, String prefijo) {
    if (disponibles.length == 0) {
        if (esCorrecta(prefijo))
            // Escribir prefijo
    } else // Caso general
        ...
}
```

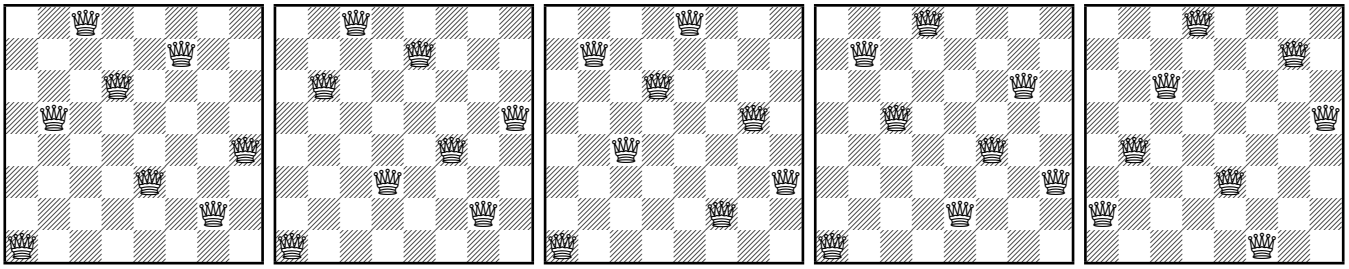
Observa que el caso general no cambia.

Si pruebas tu programa, debes obtener 92 soluciones distintas al problema. Las cinco primeras son

```
15863724
16837425
17468253
17582463
24683175
```

[1] Por ejemplo: `http://es.wikipedia.org/wiki/Problema_de_las_ocho_reinas`.

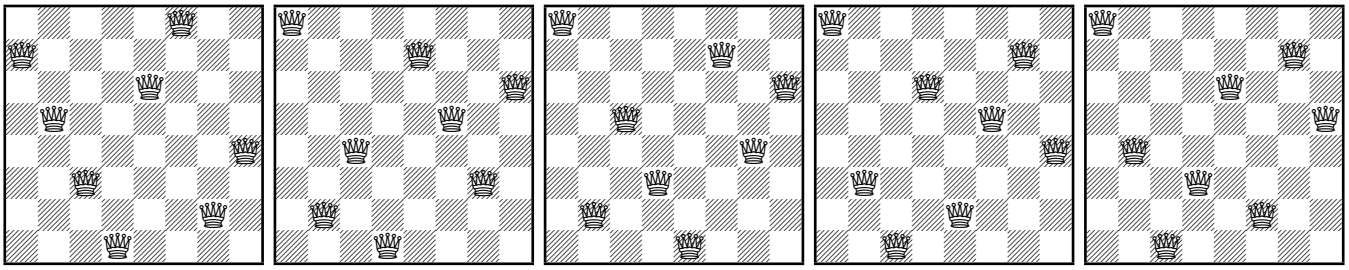
que se corresponden con estos tableros:



Las cinco últimas son

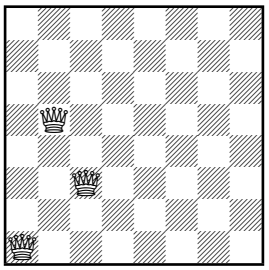
```
75316824
82417536
82531746
83162574
84136275
```

y corresponden a



Ejercicio 4

Podemos hacer que la búsqueda de las soluciones al problema de las ocho reinas sea más eficiente si nos damos cuenta de que no es posible encontrar una solución cuando el prefijo que llevamos ya tiene dos reinas atacándose. Por ejemplo, no habrá ninguna solución que comience por `153` porque las reinas de la primera y la tercera columna ya se están atacando. Gráficamente, `153` corresponde a la posición



y está claro que no hace falta que sigamos probando posibilidades.

Para implementar esta mejora, deberás hacer que `esCorrecta` acepte cadenas de longitud menor que ocho y devuelva `true` si se corresponden con un prefijo que no tenga reinas atacándose. Además, el caso general de `ochoReinas` deberá llamar a `esCorrecta` antes de la llamada recursiva de modo que sólo hará esta si `esCorrecta` ha devuelto `true`.

2. Ejercicios sobre costes

En estos ejercicios, vas a comprobar experimentalmente el coste de ejecutar una serie de programas sobre vectores de enteros. Estos problemas consistirán en hacer algún cálculo sobre un vector, por ejemplo, calcular su máximo. En cada problema, tendrás que preparar una clase que lo resuelva. Luego, utilizarás un programa de prueba que bajarás del aula virtual para resolver varias instancias del problema, cronometrar cuánto se tarda en resolverlas y mostrar gráficamente los resultados.

Tendrás que utilizar tres ficheros auxiliares: `jcommon.jar`, `jfreechar.jar` y `auxiliaresPractica4.jar`, que puedes bajar del aula virtual. Puedes descargarlos en cualquier lugar del disco, pero te aconsejamos que crees un directorio específico para ellos (por ejemplo, `auxiliaresPractica4`). Una vez los tengas, abre las propiedades del proyecto correspondiente a esta práctica, busca `Java build path/Libraries` y usa `Add external JARs` para añadirlos.

Las clases que implementes tendrán que cumplir con la interfaz `Resolvedor<Resultado>`, que está en el fichero `auxiliaresPractica4.jar` y que te obligará a implementar el método `resuelve`, que recibe un vector de enteros y devuelve un valor de tipo `Resultado`.

Ejercicio 5

Escribe la clase `EncuentraMayor` que implementa la interfaz `Resolvedor<Integer>` y cuyo método `resuelve` devuelve el máximo de los elementos del vector.

El esqueleto de la clase es:

```
public class EncuentraMayor implements Resolvedor<Integer> {
    @Override
    public Integer resuelve(int[] vector) {
        ...
    }
}
```

Calcula cuáles son los costes asintóticos en el mejor y peor caso de tu algoritmo y comprueba si se corresponden con los tiempos que obtienes.

Para hacer las mediciones y mostrar los resultados, utiliza el programa `PruebaEncuentraMayor`.

Ejercicio 6

Escribe la clase `TodosIguales` que implementa la interfaz `Resolvedor<Boolean>` y cuyo método `resuelve` devuelve `true` si todos los elementos del vector son iguales y `false` en caso contrario.

Calcula los costes en el mejor y peor de los casos.

Para hacer las mediciones y mostrar los resultados, utiliza el programa `PruebaTodosIguales`.

Ejercicio 7

Escribe la clase `TodosIgualesOrdenado`. El objetivo es mejorar los tiempos respecto al caso anterior aprovechando que los elementos del vector están ordenados. ¿Puedes cambiar `resuelve` para que tenga en cuenta que los elementos están ordenados?

Calcula los costes en el mejor y peor de los casos.

Para hacer las mediciones y mostrar los resultados, utiliza el programa `PruebaTodosIgualesOrdenado`.

Ejercicio 8

Escribe la clase `TodosDistintos` que implementa la interfaz `Resolvedor<Boolean>` y cuyo método `resuelve` devuelve `true` si todos los elementos del vector son distintos (es decir, no hay dos iguales) y `false` en caso contrario.

Calcula los costes en el mejor y peor de los casos.

Para hacer las mediciones y mostrar los resultados, utiliza el programa `PruebaTodosDistintos`.

Ejercicio 9

Repite el ejercicio anterior para la clase `TodosDistintosOrdenado`, que recibe los elementos del vector ordenados.

Nuevamente, calcula los costes, mira los tiempos y compáralos con los de `TodosDistintos`. Fíjate en los valores de los tiempos en el eje vertical.

Para hacer las mediciones y mostrar los resultados, utiliza el programa `PruebaTodosDistintosOrdenado`.

3. Ejercicios sobre búsqueda

Ejercicio 10

El fichero de texto `dniCenso.txt` contiene los DNI de las personas censadas en una determinada localidad. Por otra parte, el fichero de texto `dniClientes.txt` contiene los DNI de los clientes de una compañía de ámbito nacional. La primera línea de cada fichero contiene un valor entero que indica la cantidad de DNI que hay en ese fichero.

Nos interesa averiguar cuántas personas censadas en esa localidad son clientes de esa compañía. Sabiendo que *ninguno de los ficheros está ordenado ni contiene datos repetidos*, escribe los siguientes métodos estáticos:

- `crearVectorDni`, que reciba como parámetro el nombre de un fichero de texto con el formato descrito anteriormente y devuelva un vector de cadenas con todos los DNI que aparecen en el fichero.
- `buscarDni`, que reciba como parámetros un DNI y un vector de DNI y devuelva `true` cuando el DNI esté en el vector y `false` cuando no esté. Para ello, el método debe aplicar una búsqueda secuencial.
- `contarCoincidencias`, que reciba como parámetros dos vectores de DNI y devuelva la cantidad de DNI del primer vector que aparecen en el segundo. Para ello, debe llamar al método `buscarDni`.

En el aula virtual de la asignatura tienes disponible un método `main`[2] que crea los vectores de DNI correspondientes a los dos ficheros de texto y muestra cuántos DNI del vector del censo están en el vector de clientes. Además, se encarga de medir el tiempo que tarda en ejecutarse la llamada al método `contarCoincidencias`.

Expresa, empleando la notación O, el coste temporal de los métodos `buscarDni` y `contarCoincidencias`.

Ejercicio 11

Modifica una copia de tu solución al ejercicio anterior para que:

- El método `main` utilice el fichero `dniClientesOrdenado.txt` en lugar del fichero `dniClientes.txt`. Este nuevo fichero contiene los mismos datos que el anterior pero, como su nombre indica, su contenido está ordenado lexicográficamente de menor a mayor DNI.
- El método `buscarDni` realice una búsqueda binaria, teniendo en cuenta que los datos del vector de clientes están ordenados.

[2] Mientras desarrollas los métodos necesarios, te recomendamos utilizar ficheros y/o vectores similares a los descritos, pero con solo unos pocos datos. Una vez que hayas comprobado que todo funciona correctamente, utiliza los ficheros de datos que tienes disponibles en el aula virtual. Si todo es correcto, tu programa debe indicar que 434 personas censadas en esa localidad son clientes de esa compañía.

Expresa, empleando la notación O, el coste temporal de los métodos `buscarDni` y `contarCoincidencias`.

Ejercicio 12

Modifica una copia de tu solución al ejercicio anterior para que:

- El método `main` utilice el fichero `dniCensoOrdenado.txt` en lugar del fichero `dniCenso.txt`. Este nuevo fichero contiene los mismos datos que el anterior pero, como su nombre indica, su contenido está ordenado lexicográficamente de menor a mayor DNI.
- Se elimine el método `buscarDni`.
- El método `contarCoincidencias` tenga en cuenta que ambos vectores están ordenados y, por lo tanto, calcule las coincidencias mediante una variante del algoritmo de mezcla.

Expresa, empleando la notación O, el coste temporal del nuevo método `contarCoincidencias`.