Soluciones (versión 2, corregidas erratas en ejercicio alfa-beta)

(A continuación se proporciona una idea de las soluciones a las preguntas propuestas. Hay que tener en cuenta que las respuestas no son exhaustivas, se pretende más bien de orientar sobre la respuesta que se pedía. Por otra parte, a cada ejercicio se proporciona una respuesta correcta que normalmente no es la única, por tanto, el que tu respuesta no coincida con esta no quiere decir que el ejercicio este mal)

3.

a) Pseudocódigo para calcular un determinante por adjuntos.

- Tenemos una matriz cuadrada de orden n de la cual queremos calcular su determinante.
- Sabemos calcular el determinante para matrices de orden 1, 2 y 3. Subproblema elemental.
- Sabemos convertir el cálculo del determinante en varias operaciones con determinantes de ordenes una unidad inferiores → Descomposición recursiva.

```
función det(A:matriz;orden:integer):real;
{
 if (orden=1) then
   det:= A[1,1];  // subproblema elemental
 else
   {
    sumatorio:= 0;
    for (i:= 1 hasta orden)
      {
       M= Calcular la matriz complementaria del elemento a[1,i] ;
       if impar(i) then sumatorio:= sumatorio + A[1,i] * det(M,orden-1)
                  else sumatorio:= sumatorio - A[1,i] * det(M,orden-1)
      }
    det:= sumatorio;
   }
}
```

b) Análisis de la complejidad.

Ecuación recurrente: T(n)= n·T(n-1)+O(n2)

Complejidad: O(n! · n2) → **O(n!)**

4.

- Heurístico: Dar la moneda de mayor valor posible sin que exceda a lo que nos queda por devolver.

```
function CambioMinimo(cantidadDevolver: integer,  // cantidad que hay que devolver
                      tiposMonedas: Vector,       // tipos de monedas de mayor a menor
                      cantidadMonedas: Vector     // cantidad de cada tipo de monedas
                     ):Vector;              // devolvemos monedas usadas de cada tipo
{
 // El vector tiposMonedas está ordenado de mayor a menor valor
 monedas_cambio:= vacio;
 indice_tipo= moneda de mayor valor;

 // Hasta que encontremos la solución o
 // no se puedan coger más monedas (no hay solución)
 while (cantidadDevolver>0) and (se pueda coger otra moneda) do
 {
    if (not cantidadMonedas[indice_tipo]>0) then
       indice_tipo++;
    valor_moneda= tiposMonedas[indice_tipo];
    if (cantidadDevolver-valor_moneda>=0) then
    {
      cantidadMonedas[indice_tipo]--;
```

```
      Monedas_cambio[indice_tipo]= Monedas_cambio[indice_tipo] + 1;
      cantidadDevolver= cantidadDevolver-valor_moneda;
    }
  }
  if (cantidadDevolver==0) then Voraz:= Monedas_cambio
                               else "no hay soluciones"
}
```

5.

a) Esquema general backtracking primera solución.

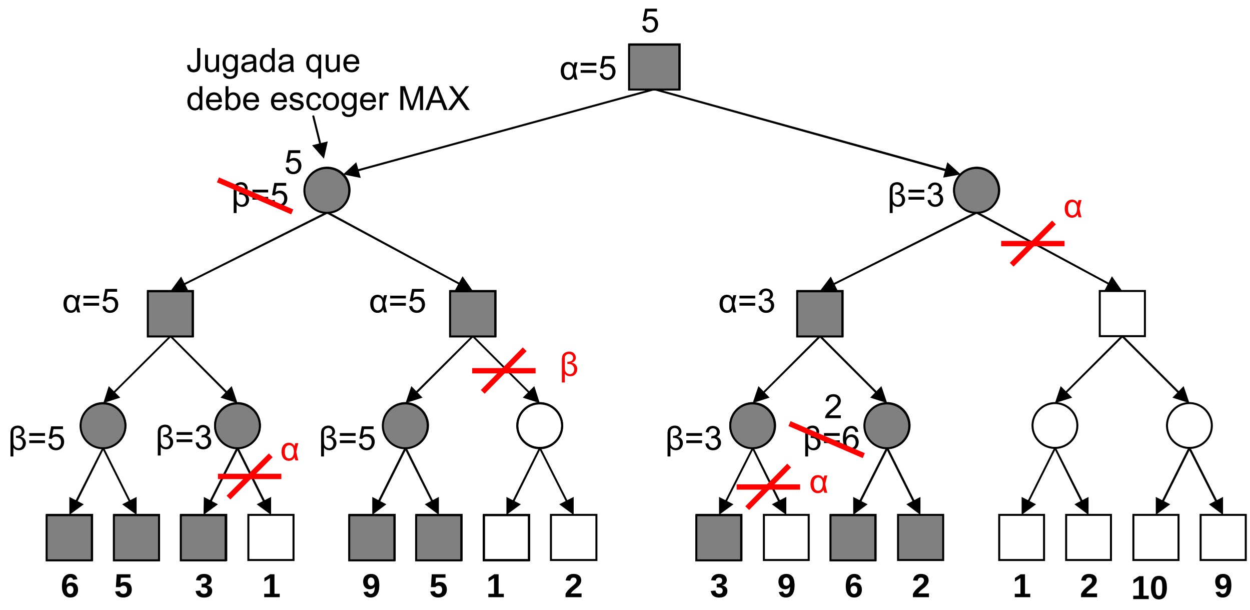
```
procedure Ensayar_nuevo_estado(estado_actual;VAR hay_solucion:boolean);
{
 saber todos los estados accesibles desde el actual;
 repeat
   seleccionar un estado;
   if estado es posible then
     begin
       anotar nuevo estado;
       if not solución final then
         begin
           Ensayar_nuevo_estado(estado,hay_solucion);
           if not hay_solucion then borrar anotación del estado;
         end
       else hay_solucion:= true;
     end
 until (no haya mas estados) or hay_solucion;
}
```

b) Método Java, que busca una solución.

```
  /**
   * Método de vuelta atrá para buscar la salida al laberinto
   */
 public boolean buscarSalida(int x, int y)
 {
   int k= -1;
   boolean haySolucion= false;
   int nx, ny; // nuevas coordenadas

   do
   {
     k++;
     nx= x+movHor[k]; ny= y+movVer[k];
     // comprobamos que la coordenada cumple las condiciones
     if ((nx>=0) && (nx<n) && (ny>=0) && (ny<n) && (mapa[nx][ny]==0))
     {
       mapa[nx][ny]= 2;
       if (!esSolucion(nx,ny))
       {
         //mostrarMapa(mapa);
         haySolucion= buscarSalida(nx, ny);
         if (!haySolucion) mapa[nx][ny]= 0;
       }
       else
         haySolucion= true;
     }
   }
   while (k<(numMov-1) && !haySolucion);
   return haySolucion;
 }
```

6.


5
α=5
Jugada que
debe escoger MAX
5
β=5
β=3
α
α=5
α=5
β
α=3
β=5
β=3
α
β=5
β=3
2
β=6
α
6
5
3
1
9
5
1
2
3
9
6
2
1
2
10
9