MEMORIA AED II PRACTICA 2

Nombre de los alumnos:

Ricardo Ramírez García ricardo.ramirezg@um.es
Pablo Salinas Rodriguez pablo.salinasr@um.es

Cuenta mooshak: G2_26
Contraseña : PWBqHF

Lista de problemas resueltos

Avance Rápido - G_AR:

Envío Nº167 Aceptado

Este envío fue aceptado ya que fuimos a hablar con Joaquín Cervera y le expusimos nuestras dudas acerca del error que nos daba el mooshak , y a la vista, de que nuestro problema resolvía bien el único caso que el mooshak nos decía que la respuesta era incorrecta ,nos modificó el estado del envío a aceptado.

El mooshak ejecutando nuestro código daba una salida diferente a la que nosotros ejecutamos en nuestra máquina, además cogimos la batería de pruebas y nos fuimos a la línea que supuestamente se encontraba el fallo ya que el mooshak ,compara la salidas para ver si son distintas, pero en nuestro caso nuestra salida era la misma que la que ofrecía el mooshak, por lo que llegamos a la conclusión de que era un fallo interno o simplemente desconocido , por lo que Joaquín vista de nuestros argumentos y viendo los resultados en nuestro ordenador y comprobando que el resultado era el correcto , decidió aceptar el envío de mooshak

Envío Nº166 Compile error :

Se nos olvidó quitar un cout y una variable que no usábamos al final

BackTracking - G_BT:

Envío Nº207 Aceptado

Envío Nº249 Runtime error ,no sabemos porque , hemos hecho una optimización del

Resolución de problemas

Avance rápido:

1. Pseudocódigo del algoritmo y explicación, justificando las decisiones de diseño, la utilización de las variables y las funciones básicas del esquema algorítmico.

```
avanceRapidoRecibir(void) : void
begin
      int[] validas;//elementos que aun se pueden escoger o descartar.
      Int[] solucion; //elementos que se han escogido.
      //inicializar las variables validas y solucion;
      int x; //moneda seleccionada en cada iteración;

      while (hayValidas(validas) AND NOT solucion(solucion))
      begin
            x = seleccionarRecibir();
            validas[x] = false;
            if(factibleRecibir(x)) insertarRecibir(x);
      end
      if(NOT solucion(solucion)) //imprimir imposible;
end
```

La función de avanceRapidoRecibir se encarga de ir cogiendo una a una las monedas necesarias para componer la solución, para ello utiliza las funciones que se explican a continuación.

```
hayValidas() : bool
begin
      for i = 0 to N
      begin
            if(validas[i] == true) return true;
      end
      return false;
end
```

Esta función recorre el array que contiene un true en las monedas que aún no se han cogido y un false en las que sí y devuelve true si encuentra algún false.

```
solucion(int[] sol) : bool
begin
      int cont = 0;
      for i = 0 to N
      begin
            cont = cont + sol[i] * valores[i];
      end
end
```

La función solucion recorre el array solución multiplicando el numero de monedas de cada tipo por el valor del tipo y sumandolo en una variable, finalmente se comprueba que el valor que se ha obtenido es igual al que se desea obtener, si es así se devuelve true, si no false.

```
factible(int elem,int[] sol) :
begin
      int x = 0;
      for i = 0 to N
      begin
            x = x + sol*valores[i];
      end
      if((V - x)/valores[elem] == 0) return false;
      return true;
end
```

La función factible comprueba si lo que queda para obtener el valor se puede conseguir metiendo monedas de ese tipo, y si las moneda que se quiere introducir es mayor que lo que queda para completar el valor.

```
insertarRecibir(int elem,int[] sol) :
begin
      int x = 0;
      for i = 0 to N
      begin
```

```
            x = x + sol * valores[i];
        end
        sol[elem] = (V-x)/valores[elem];
end
```

Se obtiene lo que resta hasta completar el valor, se divide entre el valor de la moneda actual y se incluye en la ranura de la posición de la array solución, el número de monedas a introducir de ese tipo.

2. Programación del algoritmo. El programa debe ir documentado, con explicación de qué es cada variable, que realiza cada función y su correspondencia con las funciones básicas del esquema algorítmico correspondiente.

--monedasDiscordia.cpp

3. Estudio teórico del tiempo de ejecución del algoritmo.

Para realizar el estudio teórico, se han analizado las siguientes funciones que conlleva a la resolución del problema con voraces:

LeerEntrada() -> Se inicializan los array , por lo que tres recorridos de un array conlleva un tiempo lineal cada uno de 0 hasta N tipos de monedas por lo que como consecuencia lleva a:

t(n) = $\sum_{i=}^{n} 3 = 3n$ t(n) $\varepsilon\ O(n)$

LiberarVariables() -> Se trata de la liberación de las variables dinámicas que se inicializan cada vez que se lee entrada por lo que , se han de liberar una vez terminado el voraz.
Se considera tiempo constante.
t(n) = 1.

Dentro de la función de avanceRapido hay estas funciones :

hayValidas() -> Esta función varía el orden con respecto al número de llamadas a esta función y del número de tipos del problema a resolver.
Llamando al número de tipos n , obtenemos que nos encontramos varios casos .
En el mejor caso es decir en el comienzo del problema , la inicializacion del vector validas[n] es en cada posición de ésta el valor es true; por lo tanto necesitará de un simple acceso al array por lo que el número de iteraciones es uno

t_m (n)= $\sum_{i=1}^{1} 1$ t_m (n) $\varepsilon\ \Omega(1)$

Para el pero caso el número de llamadas al método es igual al número de tipos de monedas por lo que se tiene que recorrer el el bucle completo

tM (n) = $\sum_{i=1}^{n} 1 = n$ tM (n) $\varepsilon\ O(n)$

Para el tiempo promedio , como he dicho anteriormente la función varía con respecto al número de llamadas al método por lo que :
si se llama dos veces necesita iterar dos veces por lo que el número de veces que se itera

es igual al número de llamadas al método , llamando al número de llamadas al método numCalls el tiempo es :

t_p (n) = $\sum_{i=1}^{numCalls} 1$ = n t_p (n) $\varepsilon\ \theta(n)$

SolucionDevolver():comprueba si es posible solucion y eso lo hace en tiempo lineal ya que ha de sumar los valores de las monedas almacenadas en el array solucionesDevolver

$$T_M(n) =+ \sum_{1}^{n}(1) = T(n)\ \varepsilon\ O(n)$$

SolucionRecibir():comprueba si es posible solucion y eso lo hace en tiempo lineal ya que ha de sumar los valores de las monedas almacenadas en el array solucionesRecibir()

$$T_M(n) =+ \sum_{1}^{n}(1) = T(n)\ \varepsilon\ O(n)$$

SelecionarDevolver(): hay dos iteraciones pero la primera pasa como en el hayValida() por lo que depende del número de llamadas a la funció por lo que nos encontramos con un mejor caso y con un peor caso .

El peor caso es cuando el número de llamadas a la función es igual al número de tipos de monedas que hay en el problema a resolver por lo que se ejecuta n iteraciones y otras n cuando entra en el segundo bucle las comprobaciones se consideran tiempo constante y no van influir en el orden final

$$T_M(n) =+ \sum_{1}^{n}(2) = TM(n)\ \varepsilon\ \Omega(n)$$

En el mejor caso es cuando nos encontramos en la situación de ser la primera llamada a la función , lo que conlleva a un tiempo constante; es decir una iteración para el primer bucle y como en el peor caso se le suma un tiempo lineal para poder seleccionar el máximo peso con el menor valor posible .

$$T_m(n) =+ \sum_{1}^{n}(1) = Tm(n)\ \varepsilon\ \Omega(n)$$

El tiempo promedio a partir de lo ya descubierto es la intersección entre los dos conjuntos es decir entre el $T_M(n)\ \varepsilon\ O(n)$ Y $T_M(n)\ \varepsilon\ .\ \Omega(n)$ que corresponde con tp (n) $\varepsilon\ \theta(n)$

SelecionarRecibir(): En cuanto al orden es lo mismo que el seleccionarDevolver(), lo que cambia el que las comprobaciones del segundo bucle son distintas pero , para el orden no afecta , por lo que llegamos a la misma conclusión

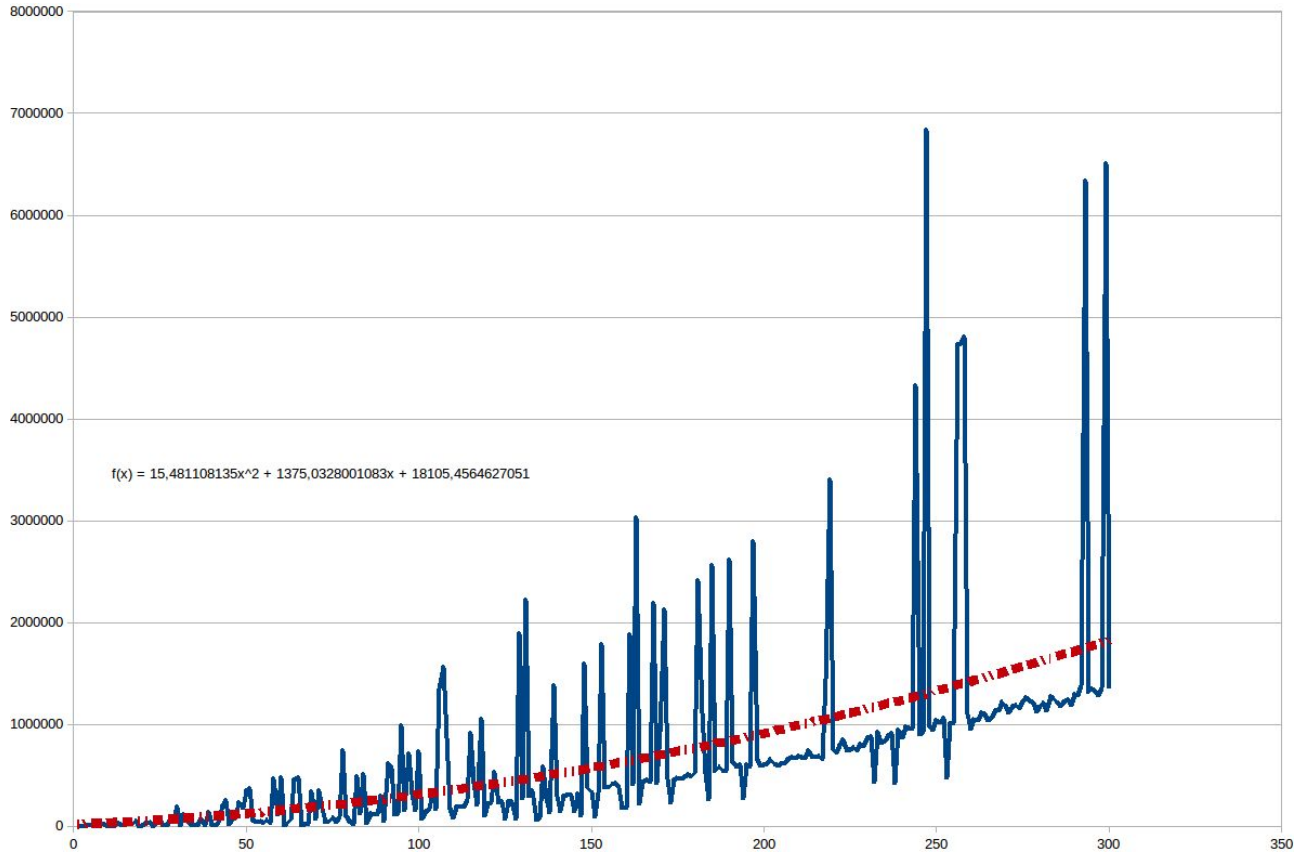
factible(): Es de orden lineal siempre recorre hasta el número de tipos por lo que simplemente el número de iteraciones del bucle es igual al número de tipos del problema (n).

$$T\ (n) =+ \sum_{1}^{n}(1) = T(n)\ \varepsilon\ O(n)$$

insertar(): Es de orden lineal siempre recorre hasta el número de tipos por lo que simplemente el número de iteraciones del bucle es igual al número de tipos del problema (n).

$$T(n) =+ \sum_{1}^{n}(1) = T(n)\ \varepsilon\ O(n)$$

4.Estudio experimental del tiempo de ejecución del algoritmo.
Con una generación de casos de 300 alternando el número del valor a alcanzar
los valores de los tipos y los pesos de los mismos de form aleatoria y , lo más importante



incrementando el número de tipos , hemos llegado a la conclusión de que viendo la línea de tendencia aproxima a nuestro resultado teórico

5.Contraste de estudio teórico y experimental, buscando justificación de las discrepancias entre los dos estudios.
Para la justificación del parecido del tiempo estimado teórico y el práctico, es que en el análisis teórico nos hemos dado cuenta de que en promedio el t(n) del mejor caso y del peor caso es del mismo orden .

Backtracking:

1.Pseudocódigo del algoritmo y explicación, justificando las decisiones de diseño, la utilización de las variables y las funciones básicas del esquema algorítmico.

backTrackingDar(void) : int

```
int nivel = 0,voa = 0;
int[] soa,sol,solFinal;
//Inicializar sol, solFinal y soa;
do
      generarDar(nivel,sol);
      if((solucionDar(nivel,sol))||(pesoDar() > voa))
      begin
            voa = pesoDar();
            soa = sol;
      endif
      if (criterioDar) nivel ++;
      else
      begin
            while((not masHermanos(nivel,sol) && (nivel > -1))
            do
                  sol[nivel] = -1;
                  nivel--;
            end
      end
while(nivel > -1);
if (voa == 0) return null;
else
begin
      //contar el número de elementos que hay de cada tipo y
representar el array resultado.
      //tam se calcular obteniendo el número máximo de monedas con
el que se puede pagar.
      For i = 0 to tam
      begin
            if (soa[s] < 0) break;
            solFinal[soa[s]]++;
      end
      return solFinal;
end
```

La función backtracking se encarga de probar todas las posibles combinaciones de monedas para completar el precio requerido dependiendo del problema que haya que solucionar quedándose con el máximo peso que sumen las monedas o con el mínimo.

generar(int nivel,int[] sol) : bool

```
if(solucionesDar[nivel] == -1)
      begin
            if(nivel == 0) sol[nivel] = 0;
            else sol[nivel] = sol[nivel-1];
      end
else
      begin
       sol[nivel] = sol[nivel] + 1;
end
```

Generar, genera un nodo hermano lo que hace es que en el nivel sumando uno al elemento que está en sol[nivel]

solucion(int nivel, int[] sol) : bool

```
int cont = 0;
for i = 0 to nivel cont = cont + valores[sol[i]];
//siendo v una variable global que indica el número que se quiere devolver.
return (cont == v);
```

Comprueba si la solucion obtenida hasta el momento es correcta.

peso(int[] sol) : int

```
int cont = 0;
for i = 0 to tam if(sol[i] > 0) cont = cont + pesos[sol[i]];
return cont;
```

Devuelve el peso de la solución hasta el momento.

criterio(nivel,int[] sol) : bool

```
return ((nivel < tam)&&(valor()+min(sol[nivel] <= v));
```

Devuelve si puede continuar a un nuevo nivel o si regresar al padre.

masHermanos(nivel,int[] sol) : bool

```
//Siendo N el número de tipos del ejercicio
return (sol[nivel]+1 < N);
```

Comprueba si hay más hermanos a los que acceder.

copiarArray(int[] solución) : int[]

```
int* aux = new int[tam];
for i = o to tam aux [i] = fuente[i];
return aux;
```

Copia una array en otra elemento a elemento.

2. Programación del algoritmo. El programa debe ir documentado, con explicación de qué es cada variable, que realiza cada función y su correspondencia con las funciones básicas del esquema algorítmico correspondiente.

--monedasMarinas.cpp

3. Estudio teórico del tiempo de ejecución del algoritmo.

Primero vamos a analizar una por una todas las funciones que forman parte del algoritmo de backtracking:

generar()-> En esta función no hay ningún bucle y las operaciones que se realizan son de coste lineal por lo tanto tanto el Tiempo en el peor caso, el tiempo en el mejor caso y el tiempo promedio de esta función son constantes.

$$T_M(n) \ \varepsilon \ O(1)$$

solucion(int nivel)-> La función solucion realiza una instrucción para declarar un contador, un for y un return por lo tanto:

$$T_M(n) = 2 + \sum_{1}^{nivel}(1) = 2 + nivel \ \varepsilon \ O(nivel)$$

Por simplicidad vamos a suponer que es el tiempo en el caso de que el nivel sea último y por tanto el tiempo será :

$$T_M(a) = \varepsilon \ O(a)$$

peso()-> La función solucion realiza una instrucción para declarar un contador, un for y un return por lo tanto:

$$T_M(n) = 2 + \sum_{1}^{n}(1) = 2 + n \ \varepsilon \ O(n)$$

criterio()-> criterio es una sola instrucción que devuelve el resultado de una comparación. Por lo tanto constante.

$$T(n) \ \varepsilon \ O(1)$$

masHermanos()-> masHermanos es una sola instrucción que devuelve el resultado de una comparación. Por lo tanto constante.

$$T(n) \ \varepsilon \ O(1)$$

copiarArray()-> La función solucion realiza una instrucción para declarar un nuevo array, un for y un return por lo tanto:

$$T_M(n) = 2 + \sum_{1}^{n}(1) = 2 + n \ \varepsilon \ O(n)$$

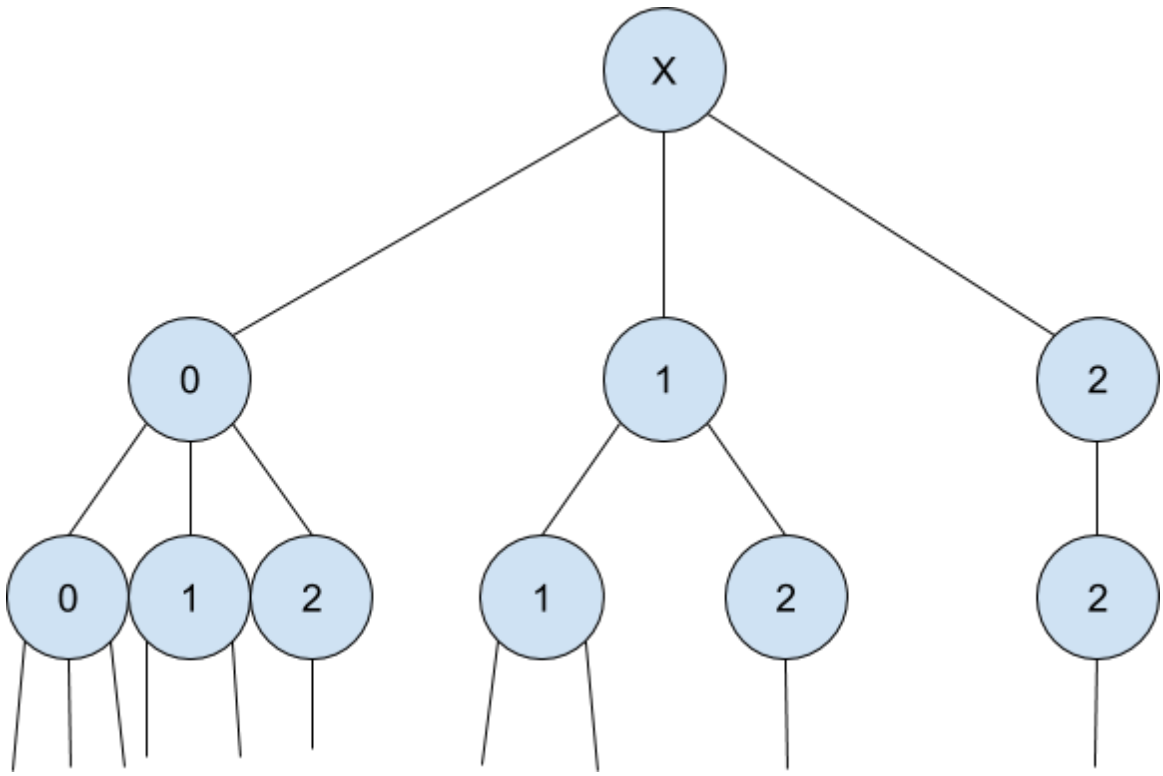
backTrackingDar(void)-> Por último en la función principal se utilizan todas las instrucciones anteriores.

Antes de comenzar a analizar el algoritmos es importante el análisis del árbol que resuelve nuestro problema ya que si calculamos el número de nodos del árbol podremos calcular el tiempo de ejecución del algoritmo calculando lo que tarda el bucle que recorre en hacer las operaciones sobre un nodo y multiplicarlo por el número de nodos ya que solo pasa una vez por cada nodo para hacer operaciones complejas, si pasa otras veces

las operaciones que realiza no tienen relevancia en el tiempo de ejecución.
Nuestro árbol se define mediante dos parámetros:

- El número de tipos de monedas que en el árbol se representa como el número de ramas que parten de un nodo en cada nivel.
- El número de niveles que se calcula dependiendo del problema, se calcula obteniendo el número de objetos que se van a coger como mucho para eso se hace V que es el valor que se desea devolver entre la moneda con menor valor.

Si hacemos un árbol que no repita casos no salen n^a nodos



Siendo n el número de tipos, en el caso del ejemplo 3 y a el valor a obtener / el valor de la moneda con menos valor.
Esto es un cálculo aproximado de nodos ya que cuando podamos se eliminan muchos nodos que tienen que ver con la entrada y no podemos controlar en el estudio por lo que podemos decir que este resultado es una cota superior ya que puede tener menos nodos pero, nunca tendrá más.

El algoritmo se basa en 2 instrucciones para declarar elementos y un for para inicializar los vectores necesarios.

$$2 + \sum_{1}^{n}(1)$$

Para analizar el bucle principal del backtraking nos ponemos como base estas consideraciones:
a(siendo **a** el número de niveles máximos del árbol, que se calcula en nuestro problema dividiendo el valor / el valor min de todos los tipos de monedas)y **n**(el número de tipos de monedas) y n^a (se considera el número de nodos ,para nosotros en una cota superior de nodos , es una aproximación ya que nuestro problema varía en función de a y el número

de tipos y no podemos establecer para un número de tipo dados el número de nodos del árbol , por eso se aproxima por n^a

En el peor caso sería generarDar(): que es de orden constante + solucionDar()--> que se ejecuta **a** veces y devuelve falso después comprueba pesoDar() ---> que es **a** veces de nuevo y la condición se cumple y se vuelve a ejecutar pesoDar() ---> que es **a** y se ejecuta también copiarArray → que vuelve a ser **a** iteraciones y todas estas operaciones se repiten por el número de nodos que es n^a **+** $2n^a$ qué son las dos instrucciones dentro del la condición de que no haya más hermanos y que el nivel sea mayor a -1 , además fuera del do while hay un bucle más que en el peor caso se ejecuta **a** veces

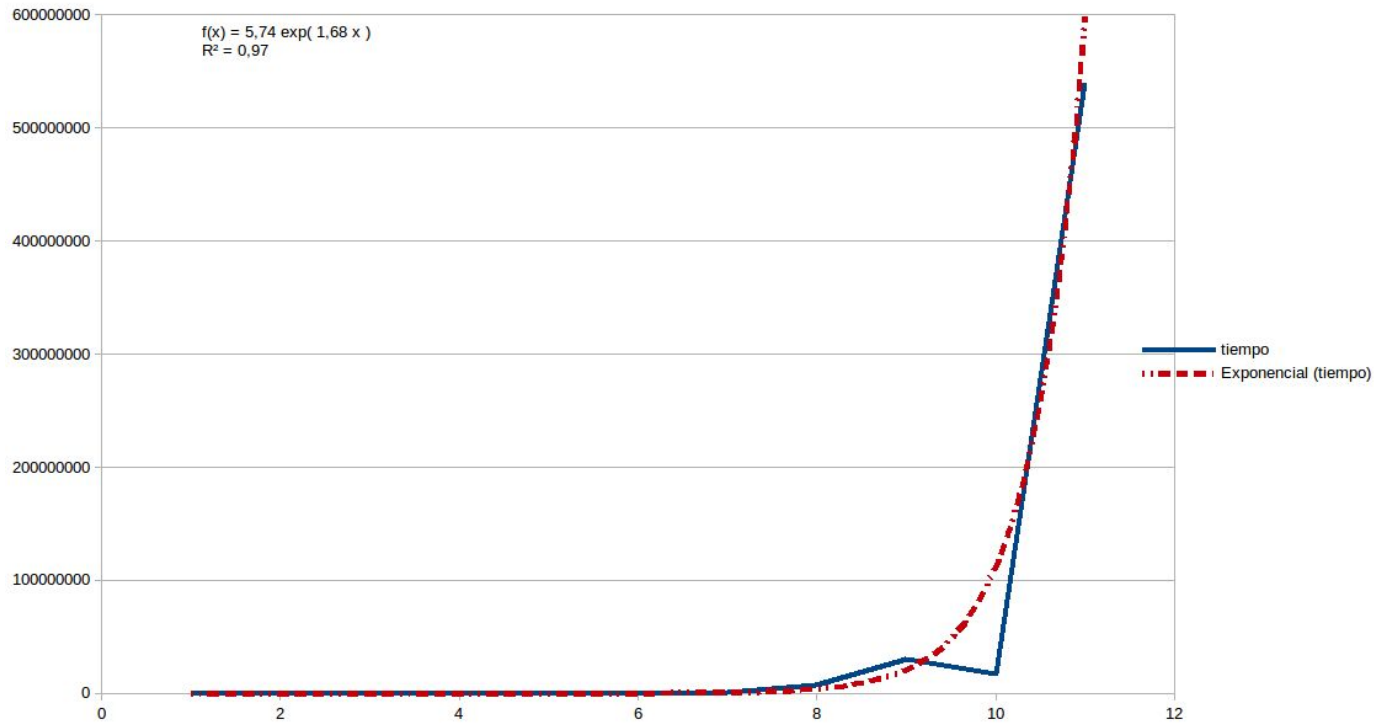
$$T_M(n) \sum_{1}^{n^a}(4a) + 2 + \sum_{1}^{n^a}(a) = 5a * n^a + 3n^a \quad \varepsilon\, O(a * n^a)$$

En el mejor caso la diferencia es que te ahorras a la hora de analizar los if dos bucles menos que no afectan al orden anterior

$$T_m(n) \sum_{1}^{n^a}(2a) + 2 = 2a * n^a + 2n^a \quad \varepsilon\, \Omega(a * n^a)$$

El tiempo promedio a partir de lo ya descubierto es la intersección entre los dos conjuntos es decir entre el $T_M(n)\ \varepsilon\ O(a * n^a)$ y $T_M(n)\ \ \varepsilon\,.\,\Omega(a * n^a))$ que corresponde con t$p\,(a * n^a)$ $\varepsilon\ \theta(a * n^a)$, somos conscientes que con la poda y el análisis de algoritmos se descartan muchísimos nodos

4.Estudio experimental del tiempo de ejecución del algoritmo.
El tiempo experimental en backtraking es exponencial nos hemos sorprendido al ver la linea de tendencia y ver reducido el orden considerablemente , ya que en promedio gracias a las podas se reduce el tiempo en promedio.



5.Contraste de estudio teórico y experimental, buscando justificación de las discrepancias

entre los dos estudios.

La única discrepancia entre los dos estudios está en las constantes multiplicativas , ya que en tema de órdenes pertenecen al mismo orden ,con la poda se reduce en número de nodos y en media el caso experimental baja la baja la base que el caso teórico es variante del número de tipos y aqui en promedio es la media aritmética de los tipos y el exponente de la línea de tendencia varía en función del número de tipos , hemos reflexionado que esta línea de tendencia es así ya que la poda de nuestro backtracking es dependiente del problema en particular y para ajustar la curva ha encontrado un patrón que la ajusta bastante bien.

Conclusiones.
A nuestro parecer la práctica ha sido bastante entretenida y lúdica pero la carga de prácticas ha sido demasiado , además hemos comprobado satisfactoriamente que los datos experimentales se corresponden con el estudio y las previsiones que habíamos hecho.