ALGORITMOS Y ESTRUCTURAS DE DATOS II

Práctica 2

José Miguel Sánchez Almagro:
josemiguel.sancheza@um.es

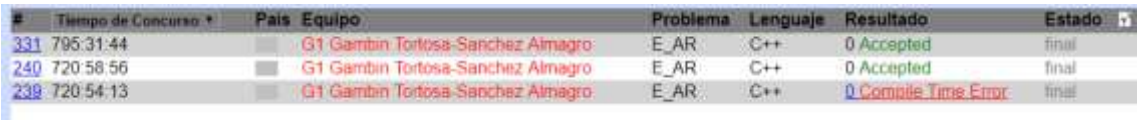
Francisco José Gambín Tortosa:
franciscojose.gambin@um.es

Cuenta Mooshak: G1_21

Grupo 1.2

LISTA DE PROBLEMAS RESUELTOS

Avance Rápido (Problema E_AR):

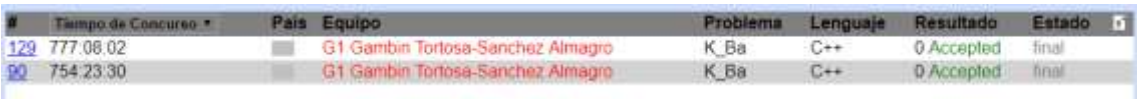


#	Tiempo de Concurso	País	Equipo	Problema	Lenguaje	Resultado	Estado
331	795:31:44		G1 Gambin Tortosa-Sanchez Almagro	E_AR	C++	0 Accepted	final
240	720:58:56		G1 Gambin Tortosa-Sanchez Almagro	E_AR	C++	0 Accepted	final
239	720:54:13		G1 Gambin Tortosa-Sanchez Almagro	E_AR	C++	0 Compile Time Error	final

El error en el primer envío se produce por una serie de warnings en el proyecto, debido a la no inicialización de las variables “averia” y “mecanico”.

El tercer envío se lleva a cabo para liberar la memoria dinámica creada en las variables R y S del main, y para comentar el código.

Backtracking (Problema K_Ba):



#	Tiempo de Concurso	País	Equipo	Problema	Lenguaje	Resultado	Estado
129	777:08:02		G1 Gambin Tortosa-Sanchez Almagro	K_Ba	C++	0 Accepted	final
90	754:23:30		G1 Gambin Tortosa-Sanchez Almagro	K_Ba	C++	0 Accepted	final

El segundo envío se lleva a cabo para liberar la memoria dinámica de la variable S creada en el main, y para la correcta documentación del código.

AVANCE RÁPIDO (PROBLEMA E_AR)

Pseudocódigo

```
voraz (R: mecanicosRestantes; var S: cjtoSolucion)
      S:= 0
      mientras (R ≠ 0) Y NO solución(S) hacer
            x:= seleccionar(R)
            R:= R - {x}
            si factible(S, x) entonces
                  insertar(S, x)
            finsi
      finmientras
```

```
solución (S: cjtoSolucion): booleano
      i: entero
      i:= 0
      mientras (i < tamaño(S)) Y (S[i] ≠ 0) hacer
            i++
      finmientras
      si i = tamaño(S) entonces
            devolver verdadero
      sino
            devolver falso
      finsi
```

```
seleccionar (R: cjtoRestantes): entero
      mejor: entero
      mecanico: entero
      i: entero
      mejor:= +∝
      i:= 0
      mientras i < tamaño(R) hacer
            si apm[i] < mejor entonces
                  mejor:= apm[i]
                  mecanico:= i
            finsi
            i++
      finmientras
      devolver mecanico
```

```
factible (S: cjtoSolucion; x: mecanico): booleano
      i: entero
      i:= 0
      mientras i < tamaño(S) hacer
            si C[i, x] = verdadero entonces
                  devolver verdadero
            finsi
      i++
```

```
      finmientras
      devolver false
```

```
insertar (S: cjtoSolucion; x: mecanico)
      i: entero
      averia: entero
      mejor: entero
      mejor:= +∞
      i:= 0
      mientras i < tamaño(S) hacer
            si (mpa[i]=0) Y (mpa[i]<mejor) Y (S[i]=0) Y
             (C[x,i]=verdadero) entonces
                  mejor:= mpa[i]
                  averia:= i
            finsi
            i++
      finmientras
      S[averia]:= x;
```

```
objetivo (S: cjtoSolucion): entero
      i: entero
      n: entero
      i:=0
      n:=0
      mientras i < tamaño(S) hacer
            si S[i] ≠ 0 entonces
                  n++
            finsi
            i++
      finmientras
devolver n
```

```
almacenar (j; entero, aux: cadenaCaracteres)
      i: entero
      x: entero
      i:= 0
      x:= 0
      mientras i < A*2 hacer
            si aux[i] = 0 entonces
                  C[j, x] = falso
            sino
                  C[j, x] = verdadero
            finsi
            x++
            i:= i + 2
      finmientras
```

```
imprimir(S: cjtoSolucion)
      para i:= 0 hasta tamaño(S) hacer
            escribir S[i]
      finpara
```

```
calcularmpa()
      n: entero
      n:= 0
      para a:= 0 hasta tamaño(S) hacer
            para m:= 0 hasta tamaño(R) hacer
                  si C[m, a] = verdadero entonces
                  n++
                  finsi
            finpara
            mpa[a] = n
            n = 0
      finpara
```

```
calcularapm()
      n: entero
      n:= 0
      para m:= 0 hasta tamaño(R) hacer
            para a:= 0 hasta tamaño(S) hacer
                  si C[m, a] = verdadero entonces
                        n++
                  finsi
            finpara
            apm[m] = n
            n = 0
      finpara
```

Expliación

En primer lugar las variables empleadas son:

- M: la cual contiene el número de mecánicos del problema.
- A: número de averías a resolver en el problema.
- C: tabla de booleanos que indica las averías que pueden ser resueltas por los mecánicos.
- S: Array de averías con cada mecánico asignado, en dicho array se encontrara la solución a nuestro problema.
- R: Array de mecánicos que indican los restantes por ser elegidos.
- mpa: Array de averías que indicara el número de mecánicos que pueden resolver cada avería.
- apm: Array de mecánicos que indica el número de averías que pueden ser resueltas por cada mecánico.

Las funciones empleadas son las siguientes:

- voraz: función principal del problema, que se encargara de ir seleccionando mecánicos (primero aquellos que puedan resolver un menor número de averías), los quitara de los posibles mecánicos restantes (array R), y en el caso de que pueda reparar una avería que todavía no haya sido reparada, lo insertara en el array de soluciones (array S).
- solución: función para conocer si todas las averías tienen un mecánico asignado, devolverá verdadero en el caso de que todas las averías tengan asignado un mecánico y falso en caso contrario.
- seleccionar: seleccionara del conjunto de mecánicos restantes(R), aquel que pueda reparar un menor número de averías.
- factible: función que comprobara si el mecánico que se le pasa como argumento puede reparar alguna avería que aún no haya sido resuelta.
- Insertar: añade el mecánico pasado como argumento al array solución, y se le asignara aquella avería que pueda ser resuelta por el menor número de mecánicos posible.
- objetivo: función que recorre el array de soluciones para devolver el número de averías que han podido ser resueltas.
- almacenar: función que se emplea en el main(), para poder almacenar en el array de booleanos C, los valores de entrada introducidos por el usuario en forma de unos o ceros y transformarlos a su correspondiente valor booleano (0 = false, 1 = verdadero).
- calcularmpa: método de apoyo que calcula el número de mecánicos que pueden resolver cada una de las averías.
- calcularapm: método de apoyo para calcular el número de averías que pueden ser resueltas por cada uno de los mecánicos.
- Imprimir: función para mostrar por pantalla de forma adecuada el array de soluciones S.

En conclusión el algoritmo voraz se ira ejecutando mientras tenga mecánicos para comprobar si pueden resolver alguna avería y mientras no se haya alcanzado ya una solución al problema.

De los posibles mecánicos que aún no han sido seleccionados, elegirá aquel que sepa resolver un menor número de averías, y lo insertara en la solución en el caso de que las averías que es capaz de resolver dicho mecánico no hayan sido resueltas ya.

Código

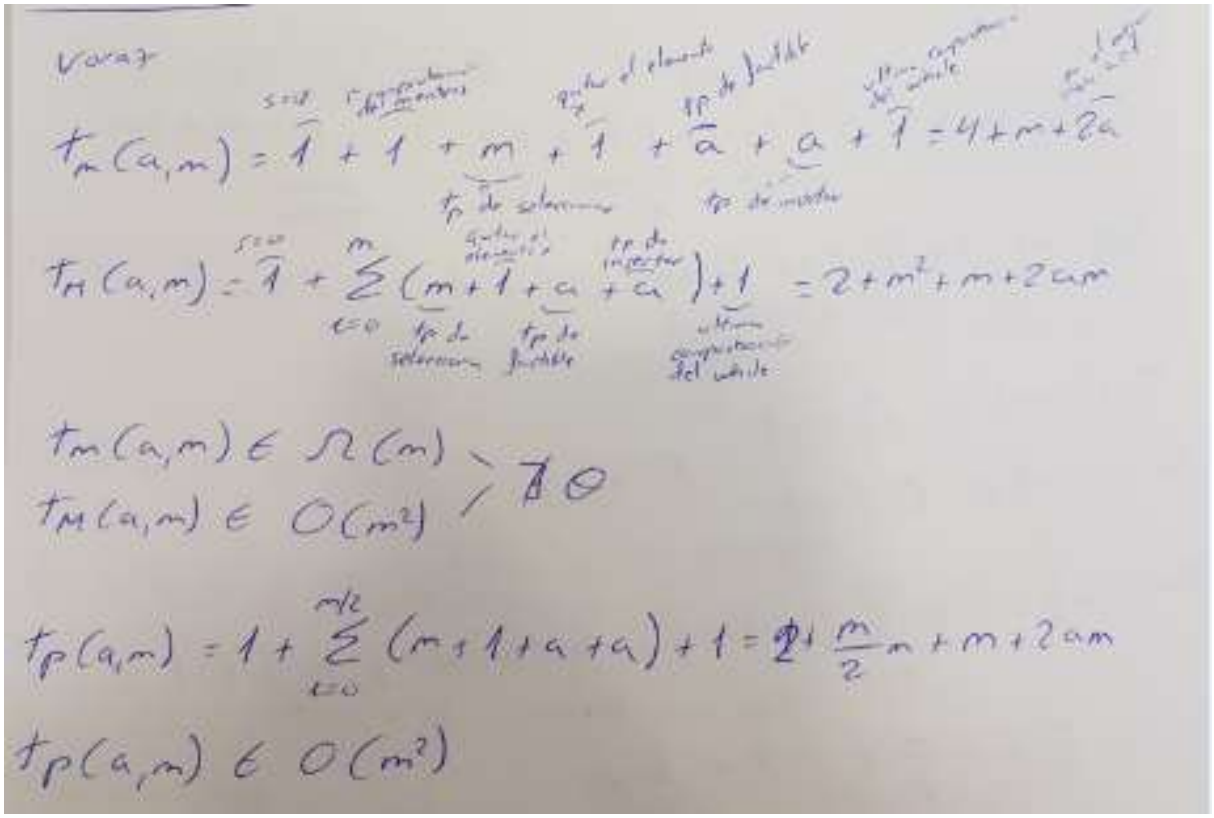
El archivo se encuentra en el fichero main.cpp

Estudio téorico

Los valores de a y m son el número de averías (A) y el número de mecánicos (M) respectivamente.

Algoritmo voraz

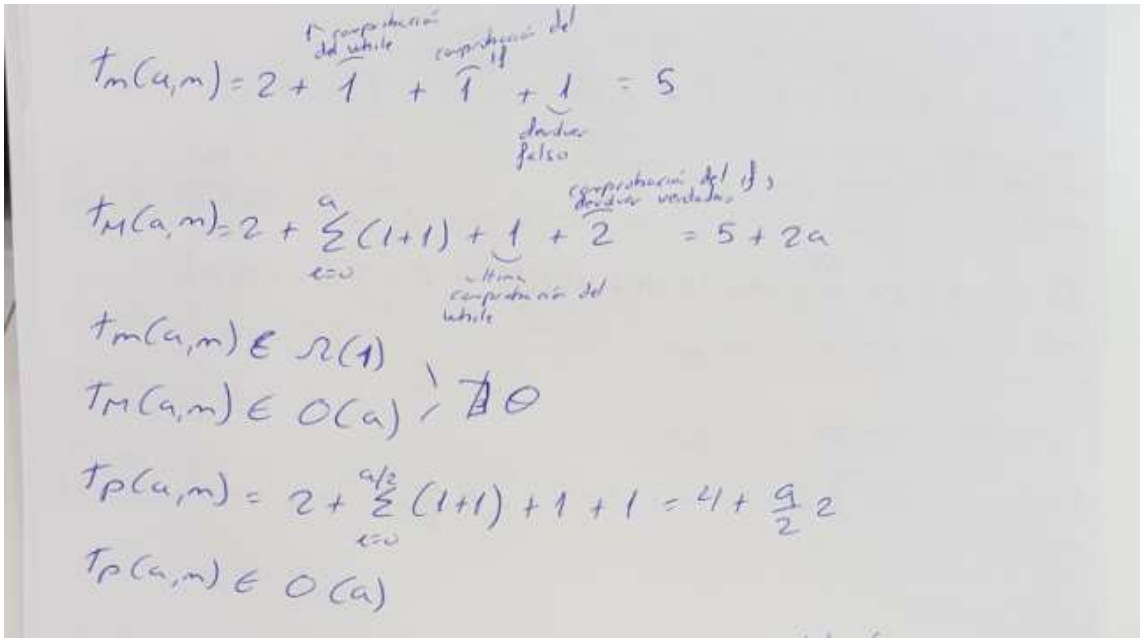
- El **mejor caso** será aquel en el que ejecuta una primera vez el bucle while completo y tras esta ejecución en la siguiente comprobación de las condiciones del while solución devuelva verdadero.
- El **peor caso** será aquel en el que todas las averías pueden ser resueltas por un mecánico.
- Por último, para la obtención del **tiempo promedio** se considerara que se llega a encontrar una solución tras recorrer la mitad de los mecánicos disponibles para reparar una avería.



Tras analizar los tiempos de ejecución se llega a la conclusión de que el peor caso empeora la eficiencia con respecto al mejor caso, esto ocurrirá muy a menudo, ya que el tiempo promedio y el peor caso tienen los mismos ordenes.

Algoritmo solución

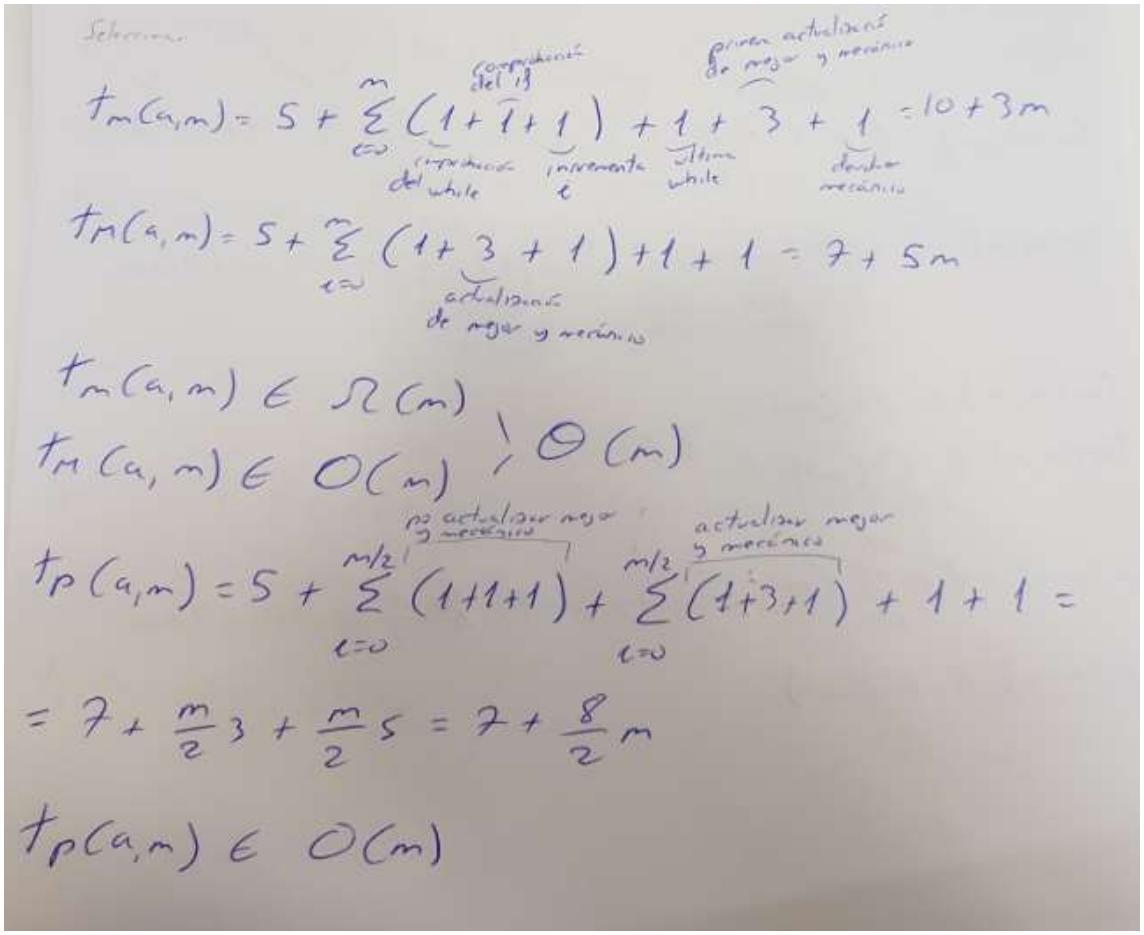
- El **mejor caso** será aquel en el que la primera avería del array de soluciones (S) no haya sido resuelta.
- El **peor caso** es aquel en el cual todas las averías han sido resueltas.
- Para el cálculo del **tiempo promedio** consideraremos que se encontrara una avería sin solución tras analizar la mitad del array de soluciones (S).



Esta función no devuelve altos tiempos de ejecución, sin embargo, el tiempo promedio y el peor caso son bastante similares, lo cual tiene dos puntos de vista: el tiempo de ejecución nunca tendrá picos altos de tiempo, y en pocas ocasiones disfrutamos de tiempos de ejecución bajos como ocurre con el mejor caso, que su tiempo es irrisorio.

Algoritmo seleccionar

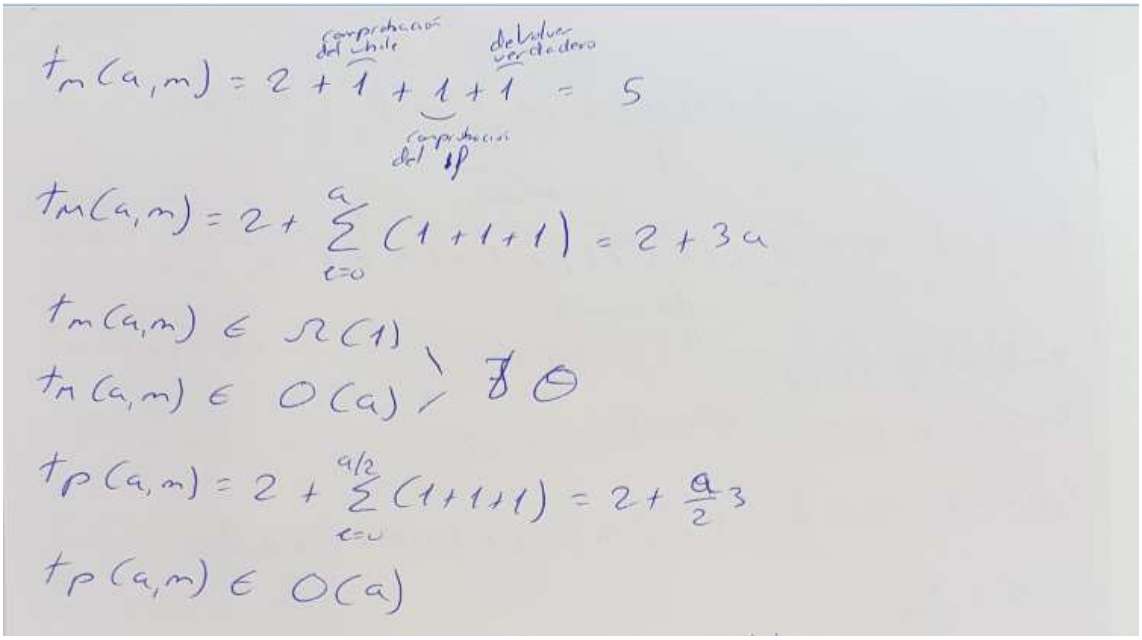
- Para el cálculo del **mejor caso** se considerara que el primer mecánico analizado será capaz de resolver un menor número de averías, respecto al resto de mecánicos.
- En el **peor caso** cada mecánico que sea analizado será capaz de resolver menos averías que el anterior mecánico estudiado.
- En el **tiempo promedio** se considerara que la mitad de los mecánicos seleccionados resolverán menos averías, respecto a los anteriores mecánicos analizados.



Llegamos a la conclusión de que se trata de una función que se comporta eficientemente puesto, ya que los órdenes para el mejor y peor caso son lineales, y por tanto tienen orden exacto de m. Por tanto será una función muy estable.

Algoritmo factible

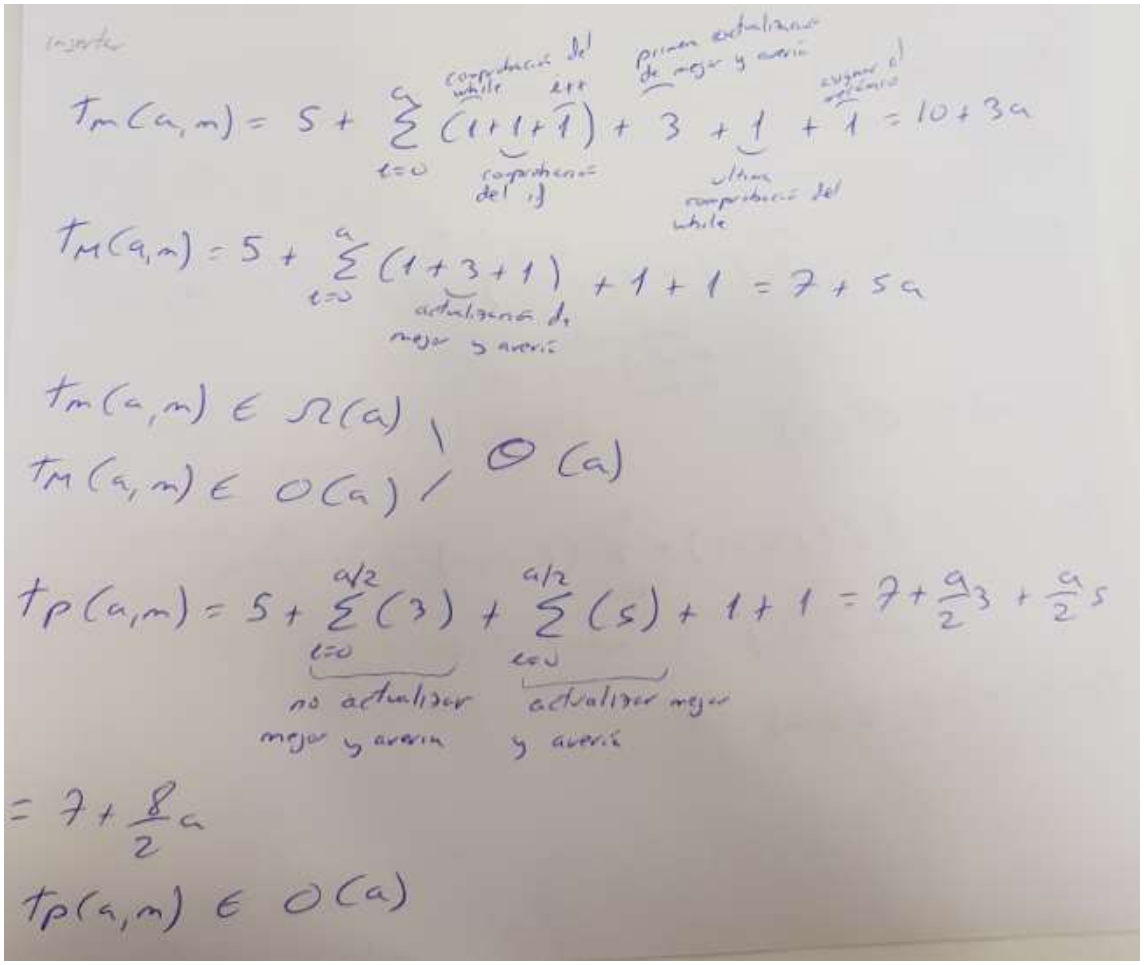
- El **mejor caso** de estudio para esta función será aquel en el que la primera avería seleccionada para analizar pueda ser resuelta por el mecánico que se le pasa como argumento a dicha función.
- En el **peor caso** el mecánico que es pasado como argumento la primera avería que podrá solucionar será la última del listado de averías.
- En el cálculo del **tiempo promedio** la primera avería que podrá solucionar el mecánico se encontrara en la mitad de las averías disponibles para solucionar.



Esta función se comportara muy eficientemente en los mejores casos, y de manera eficiente en el resto de casos, puesto que tiene un orden promedio lineal (orden a) que dependerá del número de averías que son pasadas al programa para ser resueltas.

Algoritmo insertar

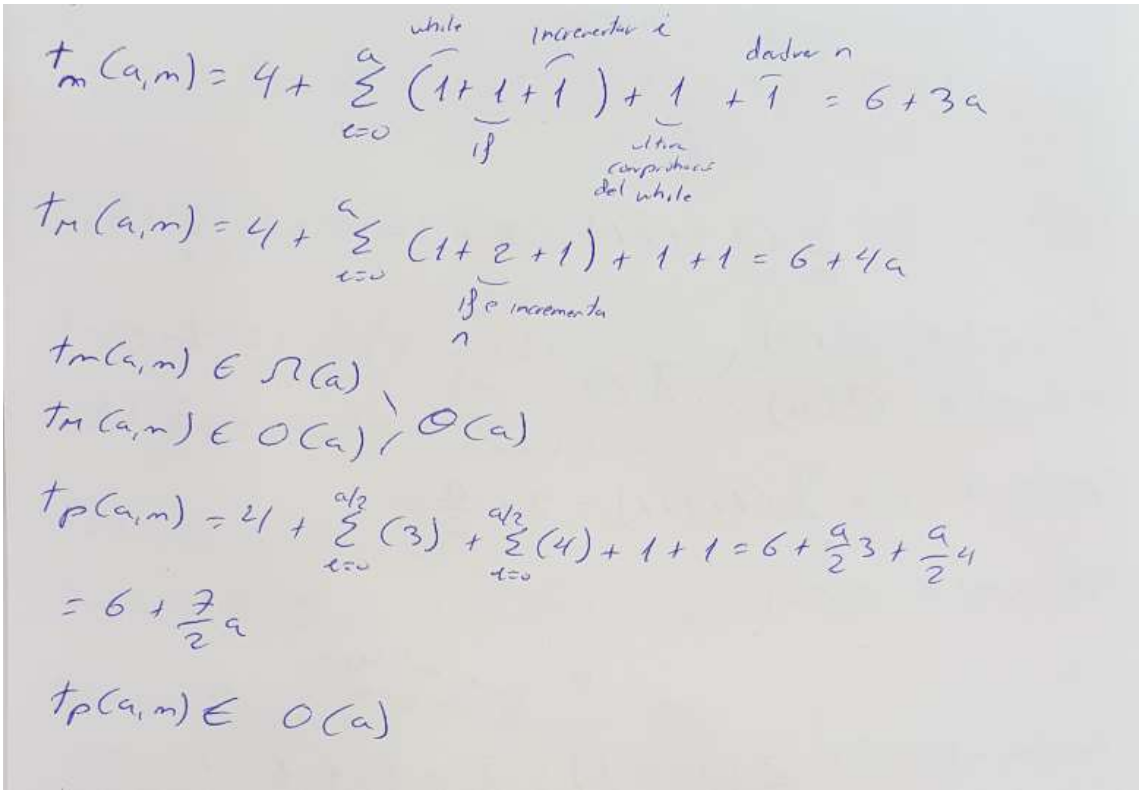
- En el **mejor caso** la primera avería será la que consideramos mejor (la avería que pueda ser resuelta por un menor número de mecánicos).
- Para el **peor caso** la mejor avería para ser seleccionada será la última que analizara la función, por tanto los valores avería y mejor se irán actualizando en cada repetición del bucle while.
- En el caso del **tiempo promedio** en esta función, que la avería que puede ser resuelta por un menor número de mecánicos se encuentra en la mitad del array mecánicos por avería (array mpa).



En conclusión podemos observar que esta función se comportara de forma muy constante puesto que sus órdenes de ejecución para el mejor y peor caso son los mismos, y por tanto dispone de un orden exacto lineal.

Algoritmo objetivo

- El **mejor caso** es aquel en el que ninguna avería ha podido ser resuelta.
- El **peor caso** será aquel en el que todas las averías han sido solucionadas.
- En el cálculo del **tiempo promedio** consideraremos que la mitad de las averías han podido ser resueltas con los mecánicos disponibles.

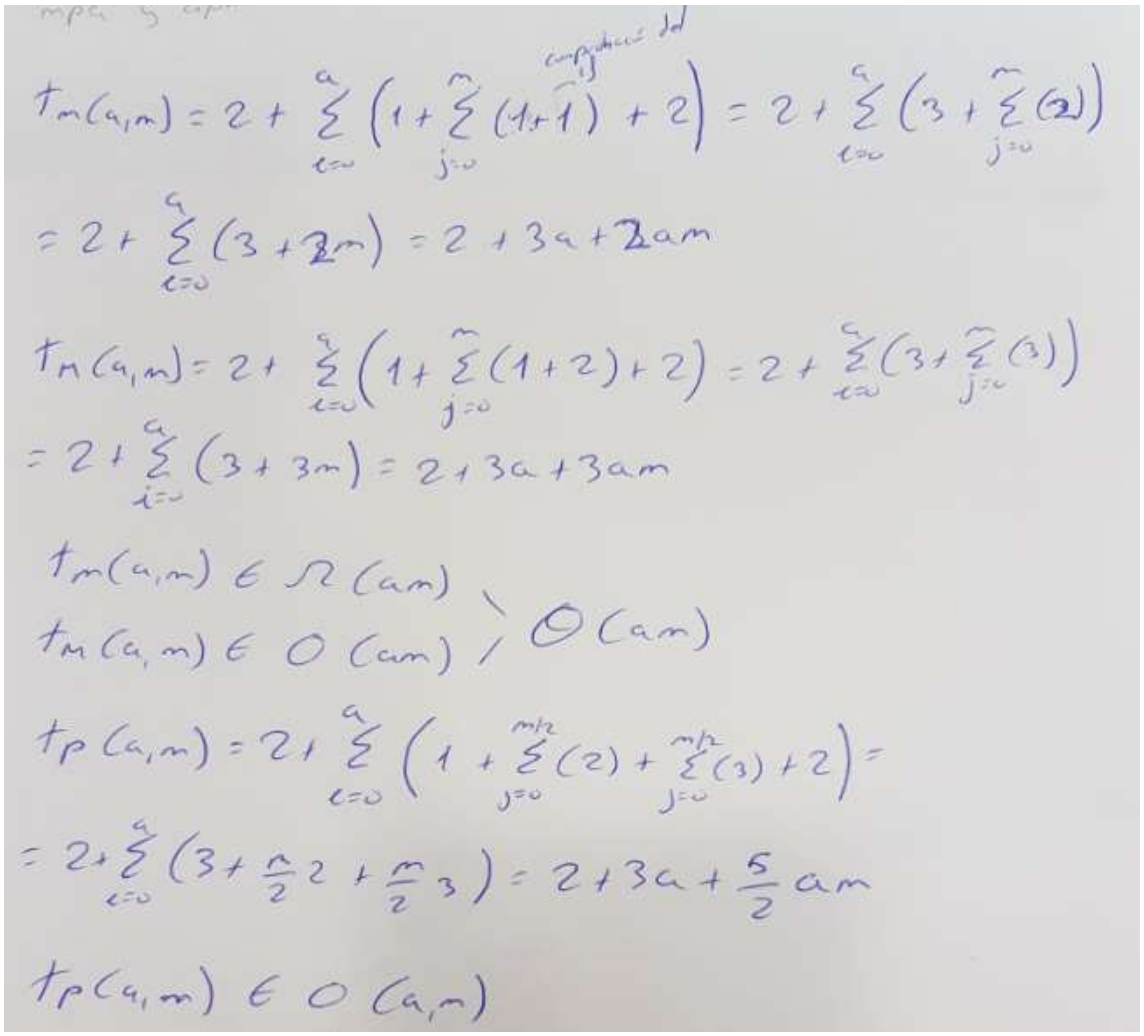


En conclusión se trata de una función eficiente con un tiempo de orden exacto lineal, y además será utilizada una sola vez en el main tras la aplicación del método voraz, por lo que no supone una función que genere un gran impacto en el programa.

Algoritmos apm y mpa

Estas dos funciones se comportan de manera prácticamente idéntica, únicamente cambia el orden de análisis del array C (primero por filas, o primero por columnas) por lo ambas serán analizadas de forma simultánea.

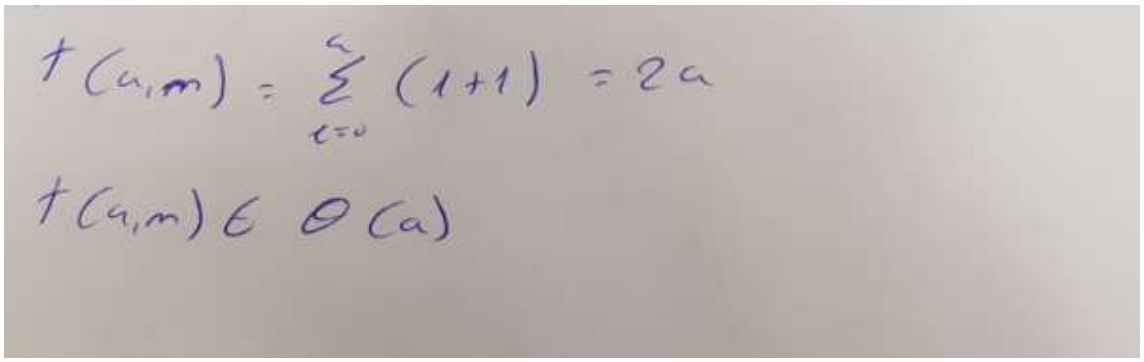
- El **mejor caso** será aquel en el que ninguna avería pueda ser resuelta por algún mecánico (o que ningún mecánico puede resolver alguna avería, apm).
- En el **peor caso** todas la averías podrán ser resueltas por cada uno de los mecánicos (o todos los mecánicos podrán resolver las averías disponibles, apm).
- En el **tiempo promedio** consideraremos que la mitad de las averías podrán ser resueltas por un mecánico (o que la mitad de mecánicos podrán resolver alguna avería).



Se trata de una función auxiliar para ahorrar cálculos a la hora de usar el resto de funciones, por lo que pese a tener una mala eficiencia, para un tamaño de a y m suficientemente grandes y parecidos se comportara con un orden cuadrático. Pese a este problema se tratara de una función que será empleada solamente una vez por cada caso de prueba disponible.

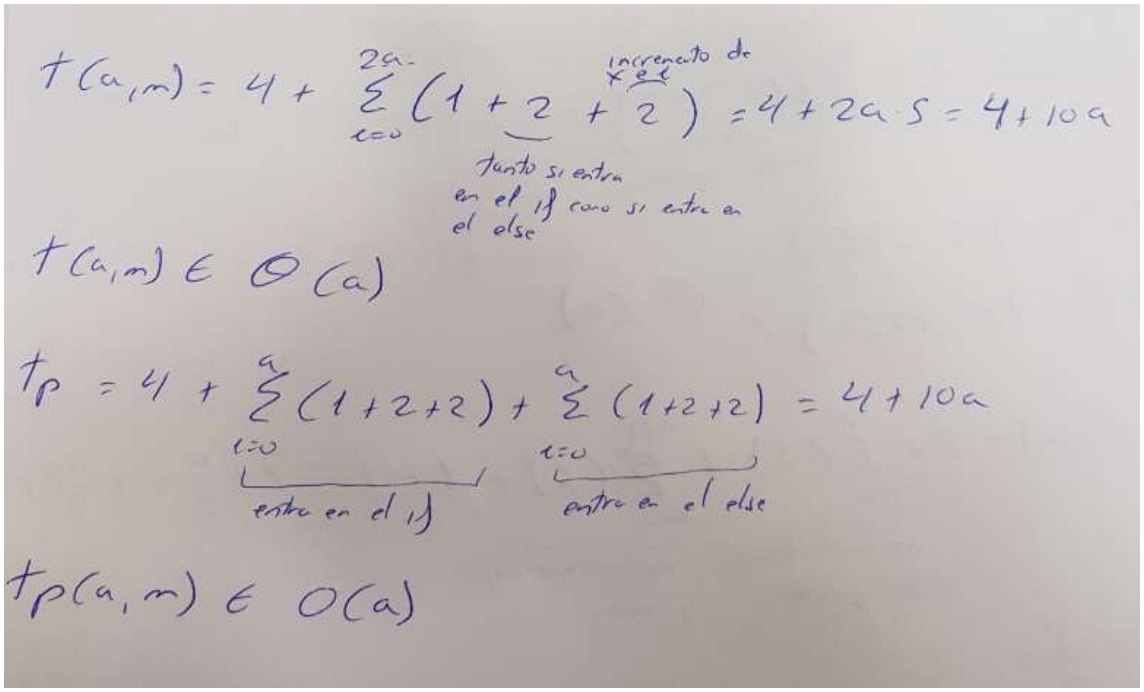
Algoritmo imprimir

Se trata de una función con apenas relevancia para la resolución del problema, empleada únicamente para la correcta devolución de los datos de salida del problema. Esta función no tiene distinción entre mejor y peor caso, ni tampoco tiempo promedio pues solo dispone de una instrucción que se ejecutara siempre.



Algoritmo almacenar

Como la anterior, es una función con apenas relevancia en la resolución del problema, cuya función será almacenar de una forma correcta los datos de entrada del problema, es una función que no dispone de distinción entre un **mejor o peor caso**. Para el cálculo del **tiempo promedio** se considerara que la mitad de los datos de entrada son 1 (el mecánico puede resolver dicha avería) y 0 (no puede resolver esta avería).



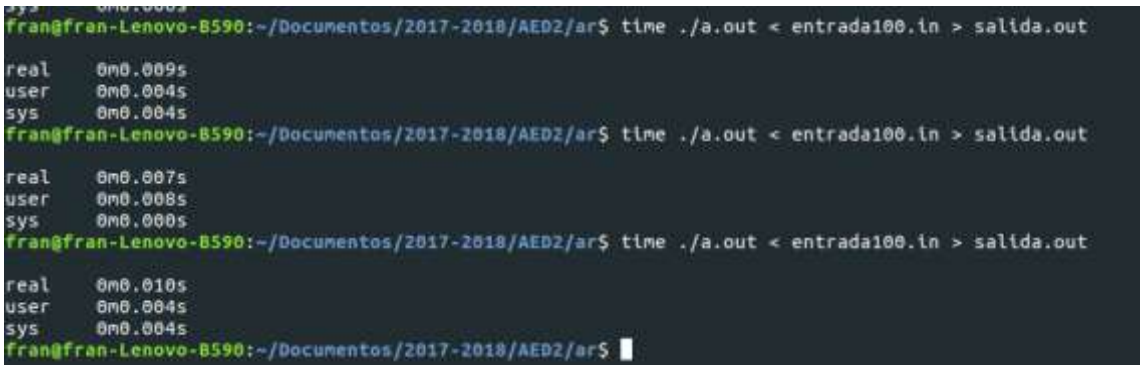
Estudio experimental

Para llevar a cabo el estudio experimental generaremos diversos archivos de entrada, en los cuales se irán incrementando el número de casos de prueba. Con estos archivos y con el uso de la orden time, mediante el terminal llevaremos a cabo diversas ejecuciones del programa y observaremos lo que ocurre con los respectivos tiempos de ejecución.

Número de casos de prueba: 50

Los resultados en las tres ejecuciones son prácticamente idénticos y rápidos, aproximadamente 9 ms.

Número de casos de prueba: 100



El tiempo con respecto al anterior experimento es prácticamente el mismo, por lo que para estos tamaños el programa se comporta de forma consistente

Número de casos de prueba: 200

```
fran@fran-Lenovo-B590:~/Documentos/2017-2018/AED2/ar$ time ./a.out < entrada200.in > salida.out

real    0m0.023s
user    0m0.016s
sys     0m0.008s
fran@fran-Lenovo-B590:~/Documentos/2017-2018/AED2/ar$ time ./a.out < entrada200.in > salida.out

real    0m0.027s
user    0m0.016s
sys     0m0.008s
fran@fran-Lenovo-B590:~/Documentos/2017-2018/AED2/ar$ time ./a.out < entrada200.in > salida.out

real    0m0.018s
user    0m0.016s
sys     0m0.000s
fran@fran-Lenovo-B590:~/Documentos/2017-2018/AED2/ar$
```

Al doblar el número de casos de prueba, con respecto al anterior experimento podemos observar que los tiempos ahora se elevan de forma considerable, perdiendo en parte la buena eficiencia para un número de casos pequeño.

Número de casos de prueba: 500

```
fran@fran-Lenovo-B590:~/Documentos/2017-2018/AED2/ar$ time ./a.out < entrada500.in > salida.out

real    0m0.045s
user    0m0.012s
sys     0m0.028s
fran@fran-Lenovo-B590:~/Documentos/2017-2018/AED2/ar$ time ./a.out < entrada500.in > salida.out

real    0m0.040s
user    0m0.024s
sys     0m0.012s
fran@fran-Lenovo-B590:~/Documentos/2017-2018/AED2/ar$ time ./a.out < entrada500.in > salida.out

real    0m0.042s
user    0m0.016s
sys     0m0.024s
fran@fran-Lenovo-B590:~/Documentos/2017-2018/AED2/ar$
```

Al incrementar en esta ocasión en más del doble el número de casos pasados al programa, podemos observar que los tiempos viéndose incrementados en proporciones del doble de tiempo respecto a la anterior ejecución.

Número de casos de prueba: 2000

```
fran@fran-Lenovo-B590:~/Documentos/2017-2018/AED2/ar$ time ./a.out < entrada2000.in > salida.out

real    0m0.118s
user    0m0.064s
sys     0m0.048s
fran@fran-Lenovo-B590:~/Documentos/2017-2018/AED2/ar$ time ./a.out < entrada2000.in > salida.out

real    0m0.106s
user    0m0.040s
sys     0m0.064s
fran@fran-Lenovo-B590:~/Documentos/2017-2018/AED2/ar$ time ./a.out < entrada2000.in > salida.out

real    0m0.098s
user    0m0.040s
sys     0m0.056s
fran@fran-Lenovo-B590:~/Documentos/2017-2018/AED2/ar$
```

En este último experimento incrementamos el número de casos de prueba, en una proporción de 4 con respecto al experimento anterior. Podemos observar que los tiempos en este experimento son prácticamente 4 veces mayores que los proporcionados en el ejercicio anterior.

En conclusión podemos observar, que nuestro algoritmo voraz funcionas eficientemente para un número de casos menor que 100. Para tamaños superiores de casos de prueba podemos observar que los tiempos de ejecución se van incrementando en la misma proporción en la que se ven incrementados los casos de prueba, lo cual puede ser explicado por el orden cuadrático en el tiempo promedio del algoritmo voraz calculado anteriormente.

Contraste entre tiempos teóricos y experimentales

Ambos estudios dejan entrever que el programa es bastante estable en tiempo promedio, a pesar de que el algoritmo voraz tiene un orden promedio cuadrático. El programa se comporta de forma bastante eficiente y rápida para un número de casos de prueba cerca de un tamaño 100.

Las principales discrepancias que podemos encontrar entre el estudio teórico y experimental pueden deberse a dos motivos principalmente.

El primero de ellos de se debe a la no consideración en el estudio teórico de factores externos al programa y sus algoritmos, como pueden ser, la maquina en la que se lleva a cabo la ejecución del problema, las tareas que está realizando el sistema operativo en el momento de la ejecución del programa, etc.

El segundo de los motivos de discrepancia entre estudio teórico y experimental, se puede encontrar en las diferencias que se plantean a la hora de plasmar en un lenguaje de programación (por ejemplo C++) con respecto a lo planteado en los algoritmos del pseudocódigo. Por ejemplo en el algoritmo voraz, a la hora de su implementación en C++ para inicializar el array de soluciones S y el array de mecánicos disponibles R, es necesario el recorrido de ambos de ambos arrays para la correcta inicialización de estos.

Esta inicialización de los array puede ser bastante costosa para tamaños elevados de las variables globales A y M (número de averías y mecánicos respectivamente) , puesto que el número de iteraciones en ambas inicializaciones dependerá de los valores de estas variables.

Backtracking (Problema K_Ba)

Pseudocódigo

El problema se ha resuelto como el de los montones. Tenemos los lugares que rellenar, por lo tanto, dividimos el peso total de los alumnos entre 2, y cada lugar tendrá ese peso como máximo. Hacemos lo propio con los alumnos.

Nuestro algoritmo intentará rellenar un montón de esos dos, intentando acercarse lo máximo posible al límite de peso, pero sin sobrepasarlo. Cuando se tenga el mejor valor, su solución contendrá los alumnos que han sido escogidos para este montón. El otro montón tendrá el resto de los alumnos que no han sido escogidos para el montón 1, y el peso de todos ellos.

Para resolver este problema con backtracking, se ha escogido el árbol binario, ya que el problema trata de escoger o no escoger un alumno a cada paso y en cada situación concreta, representado con un 1 si escogemos el alumno, y un 0 si no lo hacemos. Además, tampoco nos importa el orden de los alumnos, sino simplemente cual cojamos en cada montón.

Con la función `Sobrepasa` evaluamos si algún alumno de los que nos quedan por visitar no excede el límite de peso, o si por escoger un alumno más no excedemos el límite de alumnos. Si estas condiciones son ciertas, podemos seguir avanzando en el árbol para ver si escogemos algún alumno más. Si no se cumplen, todas las ramas que quedan por debajo no se incluirán en la solución en ningún caso, y por tanto las "podamos".

Se usarán las siguientes variables (aunque algunas no se usan en el pseudocódigo):

- **S : array de enteros** -> Contiene la solución parcial al problema en cada momento. Cada posición *i* corresponde a un alumno a_i. Si el alumno a_i ha sido escogido para el montón, S_{ai} valdrá 1; si no ha sido escogido, 0; y si no ha sido evaluado, -1.
- **nivel : entero** -> Nivel en el que estamos en cada momento en el árbol lógico (ficticio) de nuestro problema.
- **voa : entero** -> Valor óptimo actual.
- **soa : array de enteros** -> Solución óptima actual. Tiene el mismo formato que S.
- **nalumnos : entero** -> Número total de alumnos. El número máximo de alumnos es 30.
- **peso : array de enteros** -> Almacena el peso de cada alumno *i*. El peso estará entre 1 y 450.
- **pesototal : entero** -> Peso total de todos los alumnos.
- **pact : entero** -> Peso de los alumnos que hay actualmente en el montón.
- **alact : entero** -> Número de alumnos que hay actualmente en el montón.
- **límitep : entero** -> Límite de peso en el montón. Se calcula dividiendo el peso total de los alumnos entre 2.
- **límitea : entero** -> Límite de alumnos en el montón. Se calcula dividiendo el número de alumnos entre 2, y si es impar tendrá un alumno más que el otro montón, es decir, si tenemos 5 alumnos, el montón tendrá como límite de alumnos 3.

```
operacion Backtracking(var S:CjtoSolucion)

  nivel:=1;
```

```
S:=inicializar;
soa:=inicializar;
voa:=-∞;
pact:=0;
alact:=0;
mientras (nivel > 0) hacer
  Generar(S, nivel);
  si Solucion(S, nivel) AND (Valor(S) > voa) entonces
    voa:=Valor(S);
    soa:=S;
  finsi
  si Criterio(S, nivel) entonces
    nivel++;
  sino
    mientras NOT MasHermanos(S, nivel) AND (nivel > 0) hacer
      Retroceder(S, nivel);
    finmientras
  finsi
finmientras
```

```
operacion Generar(var S:CjtoSolucion, var nivel:entero)
  S_nivel := S_nivel + 1;
  pact := pact + S_nivel*peso_nivel;
  alact := alact + S_nivel;
```

```
operacion Solucion(var S:CjtoSolucion, var
nivel:entero):booleano
  devolver (pact <= limitep) AND (alact <= limitea);
```

```
operacion Criterio(var S:CjtoSolucion, var nivel:entero):booleano
  devolver Solucion(S, nivel) AND (nivel < nalumnos) AND NOT
Sobrepasa(S, nivel);
```

```
operacion Sobrepasa(var S:CjtoSolucion, var
nivel:entero):booleano
  var i:entero;
  i := nivel + 1;
  si NOT (alact = limitea) entonces
    mientras (i <= nalumnos) hacer
      si ((pact + peso_i) <= limitep) devolver FALSO;
      finsi
      i++;
    finmientras
  finsi
  devolver VERDADERO;
```

```
operacion MasHermanos(var S:CjtoSolucion, var
nivel:entero):booleano
  devolver S_nivel < 1;
```

```
operacion Retroceder(var S:CjtoSolucion, var nivel:entero)
  pact := pact - peso_nivel * S_nivel;
  alact := alact - S_nivel;
  S_nivel := -1;
  nivel--;
```

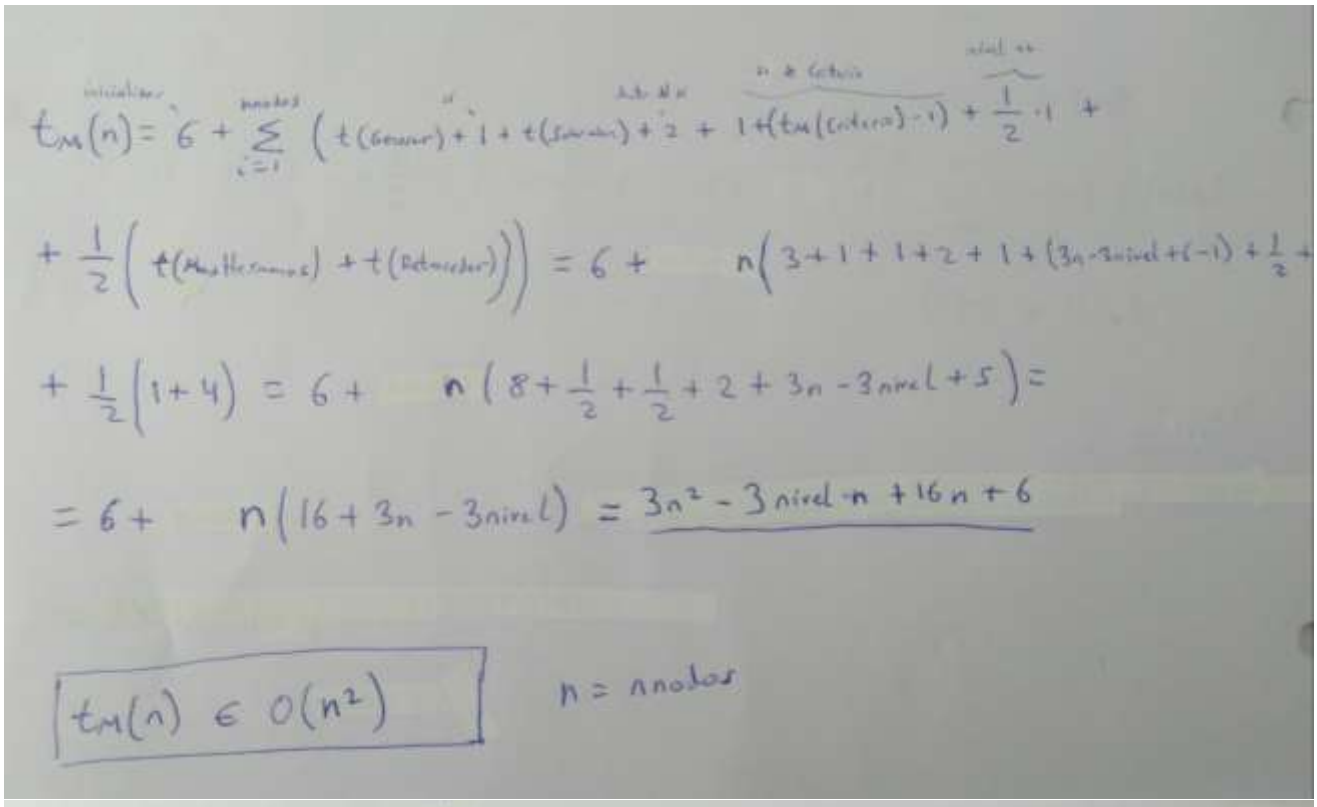
En el archivo *backtracking.cpp* se encuentra el código con el programa, ahí se encuentran estas funciones y la función `main()`, que inicializa los valores, lee e imprime por pantalla, y llama a la función `backtracking`.

Código

En el archivo *backtracking.cpp*.

Estudio teórico

Algoritmo backtracking



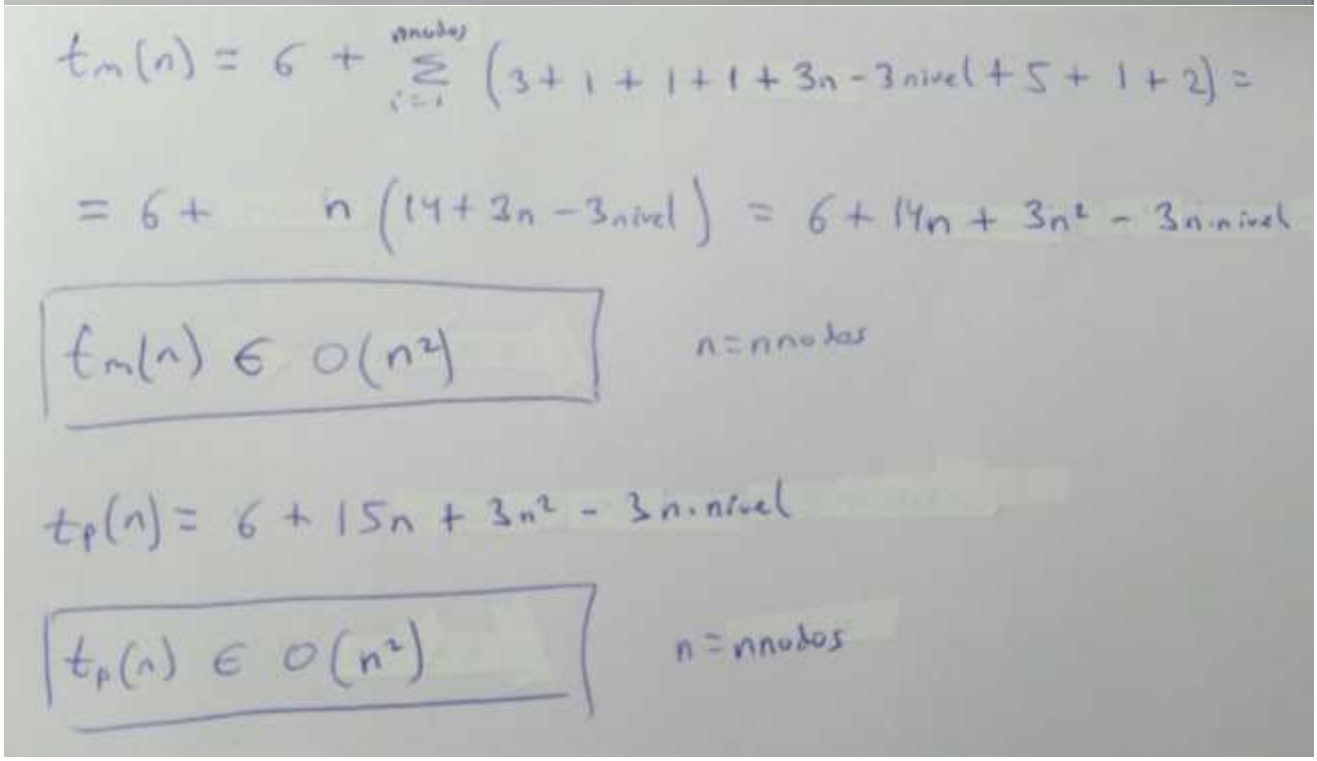
$$t_M(n) = 6 + \sum_{i=1}^{nnodos} \left(t(\text{Generar}) + 1 + t(\text{Solución}) + 2 + 1 + t_M(\text{Criterios}) \cdot (-1) + \frac{1}{2} \cdot 1 + \right.$$

$$\left. + \frac{1}{2} \left(t(\text{Marcar}) + t(\text{Retroceder}) \right) \right) = 6 + n\left(3 + 1 + 1 + 2 + 1 + (3n - 3nivel + (-1)) + \frac{1}{2} + \right.$$

$$+ \frac{1}{2}(1+4) = 6 + n\left(8 + \frac{1}{2} + \frac{1}{2} + 2 + 3n - 3nivel + 5\right) =$$

$$= 6 + n(16 + 3n - 3nivel) = 3n^2 - 3nivel \cdot n + 16n + 6$$

$t_M(n) \in O(n^2)$ $\quad n = nnodos$



$$t_m(n) = 6 + \sum_{i=1}^{nnodos} (3 + 1 + 1 + 1 + 3n - 3nivel + 5 + 1 + 2) =$$

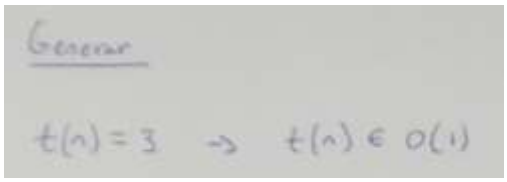
$$= 6 + n(14 + 3n - 3nivel) = 6 + 14n + 3n^2 - 3n \cdot nivel$$

$t_m(n) \in O(n^2)$ $\quad n = nnodos$

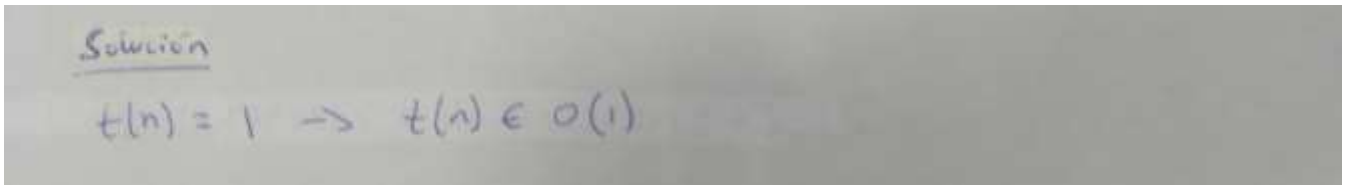
$$t_P(n) = 6 + 15n + 3n^2 - 3n \cdot nivel$$

$t_P(n) \in O(n^2)$ $\quad n = nnodos$

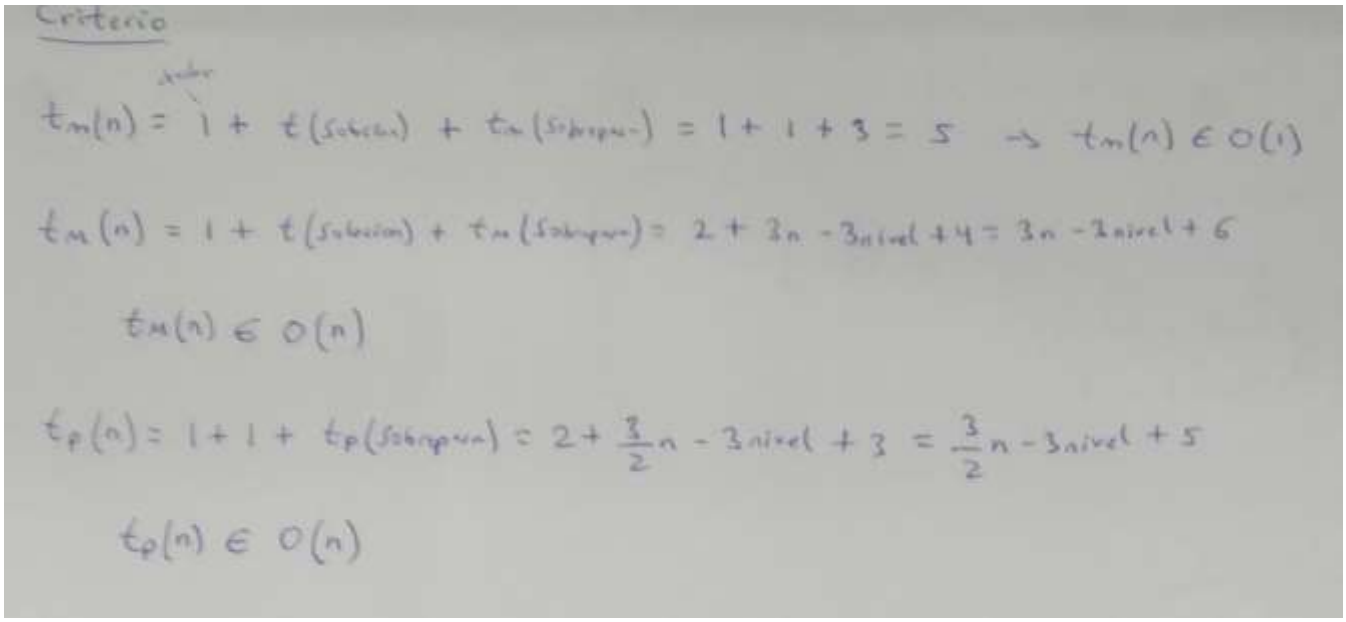
Función Generar

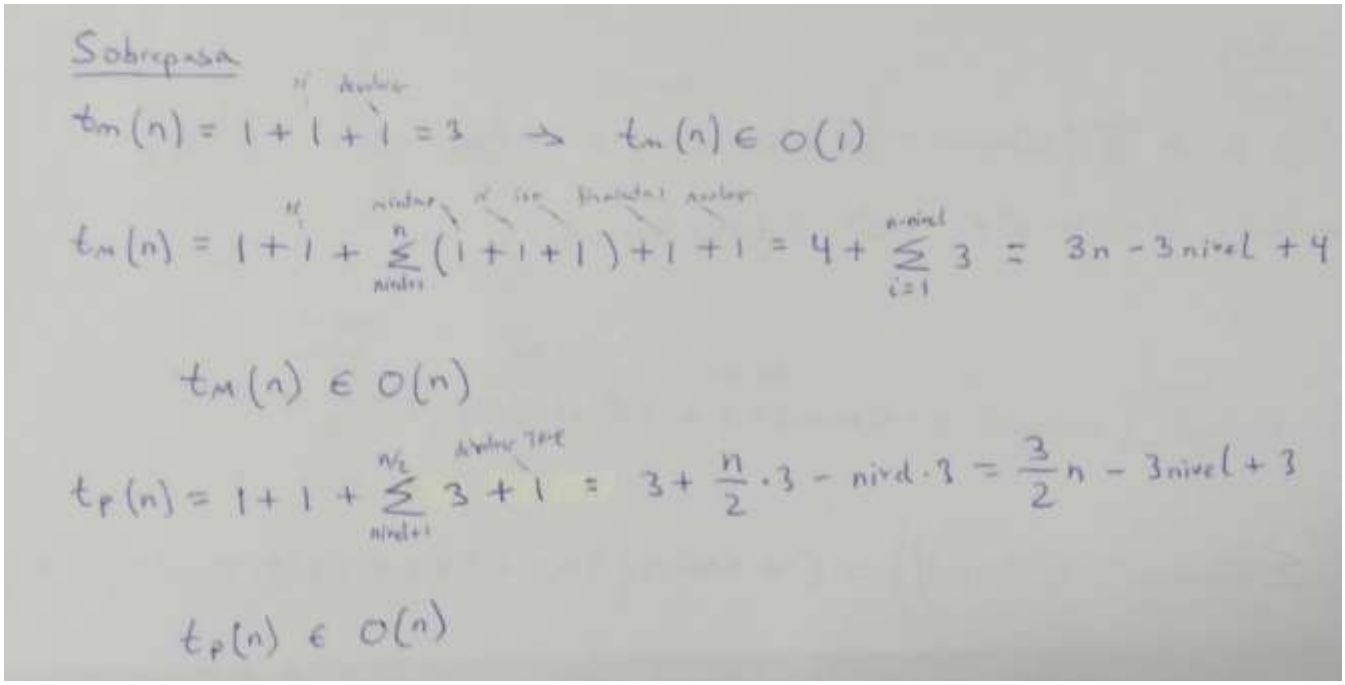


Función Solución

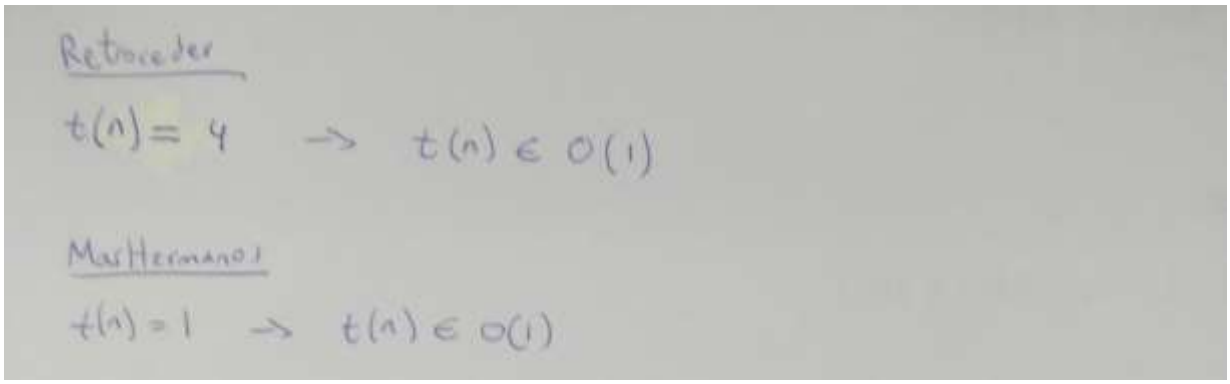


Funciones Criterio y Sobrepasa





Funciones Retroceder y MasHermanos



Estudio experimental

Mejor caso

El mejor caso está formado por un solo alumno de peso 1. Las tres ejecuciones dieron los siguientes tiempos:

```
mrchemi@MrChemi-Linux:~/0Universidad/0-ALGORITMOS_Y_ESTRUCTURAS_DE_DATOS_II/Practica/Temas_3_y_4/Entrega/bck$ time ./a.out < mejorcaso.in
0 1

real	0m0,004s
user	0m0,004s
sys	0m0,000s
mrchemi@MrChemi-Linux:~/0Universidad/0-ALGORITMOS_Y_ESTRUCTURAS_DE_DATOS_II/Practica/Temas_3_y_4/Entrega/bck$ time ./a.out < mejorcaso.in
0 1

real	0m0,004s
user	0m0,000s
sys	0m0,003s
mrchemi@MrChemi-Linux:~/0Universidad/0-ALGORITMOS_Y_ESTRUCTURAS_DE_DATOS_II/Practica/Temas_3_y_4/Entrega/bck$ time ./a.out < mejorcaso.in
0 1

real	0m0,004s
user	0m0,004s
sys	0m0,000s
```

Unos tiempos de apenas unos milisegundos, con un comportamiento muy rápido.

Casos intermedios

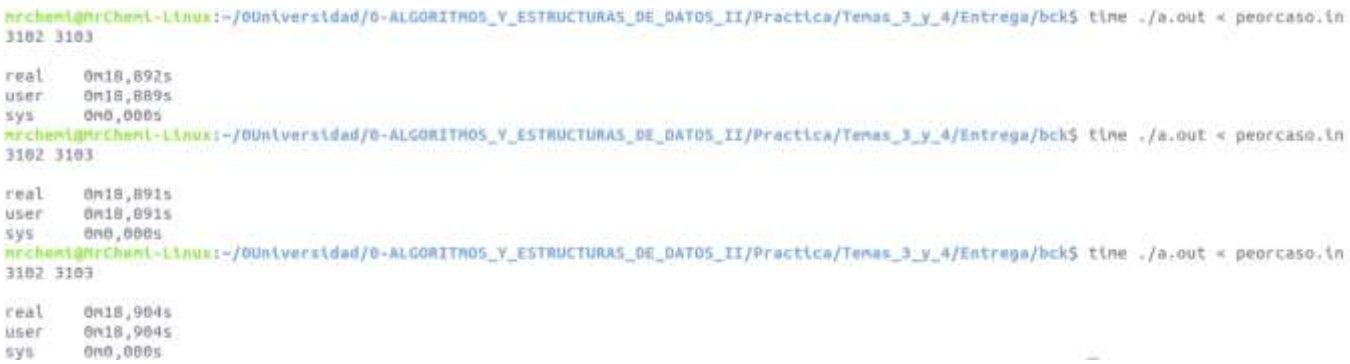
El primer caso intermedio está formado por 15 alumnos, y el segundo caso por 20 alumnos, ambos con pesos diversos en cada alumno:

```
mrchemi@MrChemi-Linux:~/0Universidad/0-ALGORITMOS_Y_ESTRUCTURAS_DE_DATOS_II/Practica/Temas_3_y_4/Entrega/bck$ time ./a.out < promedio1.in
1377 1378

real	0m0,006s
user	0m0,006s
sys	0m0,000s
mrchemi@MrChemi-Linux:~/0Universidad/0-ALGORITMOS_Y_ESTRUCTURAS_DE_DATOS_II/Practica/Temas_3_y_4/Entrega/bck$ time ./a.out < promedio1.in
1377 1378

real	0m0,006s
user	0m0,006s
sys	0m0,000s
mrchemi@MrChemi-Linux:~/0Universidad/0-ALGORITMOS_Y_ESTRUCTURAS_DE_DATOS_II/Practica/Temas_3_y_4/Entrega/bck$ time ./a.out < promedio1.in
1377 1378

real	0m0,006s
user	0m0,006s
sys	0m0,000s
mrchemi@MrChemi-Linux:~/0Universidad/0-ALGORITMOS_Y_ESTRUCTURAS_DE_DATOS_II/Practica/Temas_3_y_4/Entrega/bck$ time ./a.out < promedio2.in
1843 1843

real	0m0,026s
user	0m0,022s
sys	0m0,004s
mrchemi@MrChemi-Linux:~/0Universidad/0-ALGORITMOS_Y_ESTRUCTURAS_DE_DATOS_II/Practica/Temas_3_y_4/Entrega/bck$ time ./a.out < promedio2.in
1843 1843

real	0m0,026s
user	0m0,022s
sys	0m0,004s
mrchemi@MrChemi-Linux:~/0Universidad/0-ALGORITMOS_Y_ESTRUCTURAS_DE_DATOS_II/Practica/Temas_3_y_4/Entrega/bck$ time ./a.out < promedio2.in
1843 1843

real	0m0,025s
user	0m0,025s
sys	0m0,000s
```

Como se puede observar el cambio es significativo por tan solo 5 alumnos. Esto provoca que tengamos 2 031 616 nodos más, provocando un aumento considerable del tiempo. Por lo tanto, es difícil concretar un tiempo promedio. Lo dejaremos en 13 ms, siendo un tiempo bastante rápido.

Peor caso

El peor caso consta de 30 alumnos, con pesos diversos:



Como vemos, el tiempo ha aumentado enormemente hasta los casi 19 segundos. Esto ocurre porque el número de nodos aumenta exponencialmente y tenemos (ficticiamente) un árbol con muchos nodos y de mucha altura. Por lo tanto, el peor caso es el que peor tiempo obtiene del programa, y debe ser evitado si es posible.

Contraste entre tiempos teóricos y experimentales

Podemos observar los resultados (en número de instrucciones) que nos da el estudio teórico y ver si crecen de la misma manera que lo hace el tiempo teórico.

Mejor caso

1 alumno

Tiempo experimental = 0’004 s.

Tiempo teórico = 21 instrucciones

Casos intermedios

15 alumnos

Tiempo experimental = 0’006 s.

Tiempo teórico = 951 instrucciones

20 alumnos

Tiempo experimental = 0’026 s.

Tiempo teórico = 1566 instrucciones

Peor caso

30 alumnos

Tiempo experimental = 18’895 s.

Tiempo teórico = 27 546 instrucciones

Podemos observar un cambio grande desde el mejor caso, al caso intermedio, y un cambio muy grande del intermedio al peor caso (en número de instrucciones). Sin embargo, algo no cuadra, y es que el incremento de tiempo desde el mejor caso al caso intermedio de 15 alumnos no corresponde al incremento de instrucciones. Además, el incremento de tiempo del caso intermedio de 15 alumnos al caso de 20 es importante, mientras que el número de instrucciones ni siquiera es doblado.

Esto puede ser debido a dos motivos:

1. En el análisis teórico tenemos la variable nivel, pero en el cálculo no ha sido usada, ya que cambia mucho a lo largo del tiempo y es difícil de medir.
2. En el programa que se ejecuta en la máquina hay funciones que realizan bucles que en el pseudocódigo no existen, por lo tanto, habrá cierta variación en este sentido cuando tengamos un número de alumnos elevado.

El resto de tiempos coinciden y tienen coherencia.

Conclusiones y valoraciones personales

En general, ambos problemas fueron fácilmente planteados gracias al conocimiento teórico sobre ellos. En una primera fase hubo que plantear como iban a ser resueltos, y que tipo de algoritmo se iba a usar (función seleccionar en AR y el árbol en Backtracking, por ejemplo), y después simplemente se utilizó el esquema general de cada tipo de problema (ligeramente modificado) y se procedió a redactar las funciones en pseudocódigo.

Se encontraron mas problemas en la implementación, debido a las peculiaridades del lenguaje, errores de inicialización de variables o similares, o errores cometidos en el pseudocódigo. Pero debido a la robustez del pseudocódigo previamente redactado, se pudieron seguir unas líneas generales, y se acabaron corrigiendo los errores mostrados.

La dificultad encontrada ha sido media, y el tiempo de trabajo no ha sido mayor a 15 horas, contando la resolución de problemas y la memoria.