

**PRÁCTICA DE AVANCE RÁPIDO Y BACKTRACKING**

**Algoritmos y Estructuras de Datos II**

**Alumnos:**

Martín Piñas Ayala 23310083-C Grupo 2,1 [martin.pinas@um.es](mailto:martin.pinas@um.es)

Mario Antelo Ribera Y1106205-B Grupo 3,1 [mario.antelo@um.es](mailto:mario.antelo@um.es)

Cuenta Usada: G2\_17

Profesor: Joaquín Cervera López Correo: [jcervera@um.es](mailto:jcervera@um.es)

**Índice**

[Lista de problemas resueltos 3](#_Toc486545850)

[Avance Rápido 3](#_Toc486545851)

[Contexto 3](#_Toc486545852)

[El Problema 3](#_Toc486545853)

[Mooshak 4](#_Toc486545854)

[Backtracking 4](#_Toc486545855)

[Contexto 4](#_Toc486545856)

[El Problema 4](#_Toc486545857)

[Mooshak 4](#_Toc486545858)

[Avance rápido 5](#_Toc486545859)

[Pseudocódigo y explicación del algoritmo 5](#_Toc486545860)

[Programación del algoritmo 7](#_Toc486545861)

[Estudio teórico del tiempo de ejecución del algoritmo 12](#_Toc486545862)

[Estudio experimental del tiempo de ejecución 13](#_Toc486545863)

[Comparación del estudio teórico y el experimental 14](#_Toc486545864)

[Backtracking 15](#_Toc486545865)

[Pseudocódigo y explicación del algoritmo 15](#_Toc486545866)

[Pseudocódigo 15](#_Toc486545867)

[Programación del algoritmo 17](#_Toc486545868)

[Estudio teórico del tiempo de ejecución del algoritmo 17](#_Toc486545869)

[Estudio experimental del tiempo de ejecución 17](#_Toc486545870)

[Comparación del estudio teórico y el experimental 17](#_Toc486545871)

# Lista de problemas resueltos

Los ejercicios asignados después de usar la fórmula dada:

(1106205+ 23310083 módulo 25) + 1 = 14.

14 I\_AR + A\_Ba

Por equivocación hicimos el ejercicio I de Backtracking.

## Avance Rápido

El ejercicio I de Avance Rápido corresponde con el enunciado de “Organización en Clase” cuyo objetivo es el emparejamiento de alumnos consiguiendo el máximo beneficio.

### Contexto

La asignación de los puestos a los alumnos en una clase es uno de los problemas más complicados a los que se enfrentan los profesores de secundaria, pues hay que tener en cuenta las relaciones personales de los alumnos (la amistad entre ellos) pero también la compenetración en el trabajo. En este problema se trata de agrupar los alumnos de dos en dos, pues suponemos que los pupitres son de dos plazas. No tendremos en cuenta la posición de los pupitres en la clase y supondremos que lo único que se pretende es agrupar a los alumnos de dos en dos (si el número es impar uno quedará sólo en un pupitre) intentando maximizar la amistad y al mismo tiempo la compenetración en el trabajo de los alumnos que sentamos juntos.

### El Problema

Escribir un programa que realice una agrupación de alumnos de dos en dos maximizando la suma de los productos del grado de amistad y la compenetración en el trabajo de alumnos que se agrupan juntos. Se dispone para ello de una matriz de *amistad*y otra de *trabajo*, donde se guardan los grados de amistad y de compenetración, que pueden no ser recíprocos, por lo que las matrices no son simétricas. La amistad y la compenetración de un alumno consigo mismo se almacenarán como cero. En cada pupitre se suman los grados de amistad de los alumnos en el pupitre, y se suman los grados de compenetración, y se multiplican los resultados de las dos sumas. El valor de beneficio es la suma de los productos obtenidos en todos los pupitres (sin incluir el posible pupitre donde sólo haya un alumno).

Suponer, por ejemplo, un caso donde el número de alumnos N = 3. Sean las matrices de amistad y trabajo las siguientes:

Amistad Trabajo

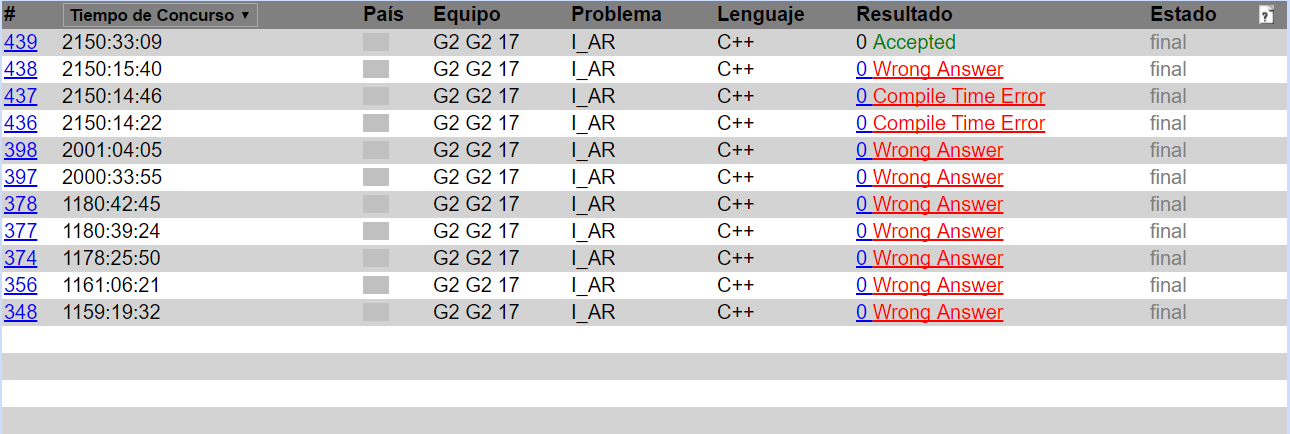
056 053  
403 302  
210 150

Si numeramos los alumnos con 0, 1 y 2, la agrupación <0, 1> tendrá valor 72 (que sale de: (5+4)\*(5+3)); la <0, 2> valor 32; y la <1, 2> valor 28, por lo que la mejor opción es sentar a los alumnos 0 y 1 juntos y dejar al 2 sólo.

Habrá que resolver el problema con avance rápido. La solución no tiene por qué ser la óptima, y se admitirán soluciones que no empeoren en más de un 30% de las obtenidas con el programa de los profesores (en el ejemplo la única admisible será la de la agrupación <0, 1>).

Se considerará que el número máximo de alumnos es 100.

### Mooshak



## Backtracking

En el caso del ejercicio I de Backtracking, corresponde con “la asignación de Averías a Mecánicos”, intentando maximizar la asignación de las averías.

### Contexto

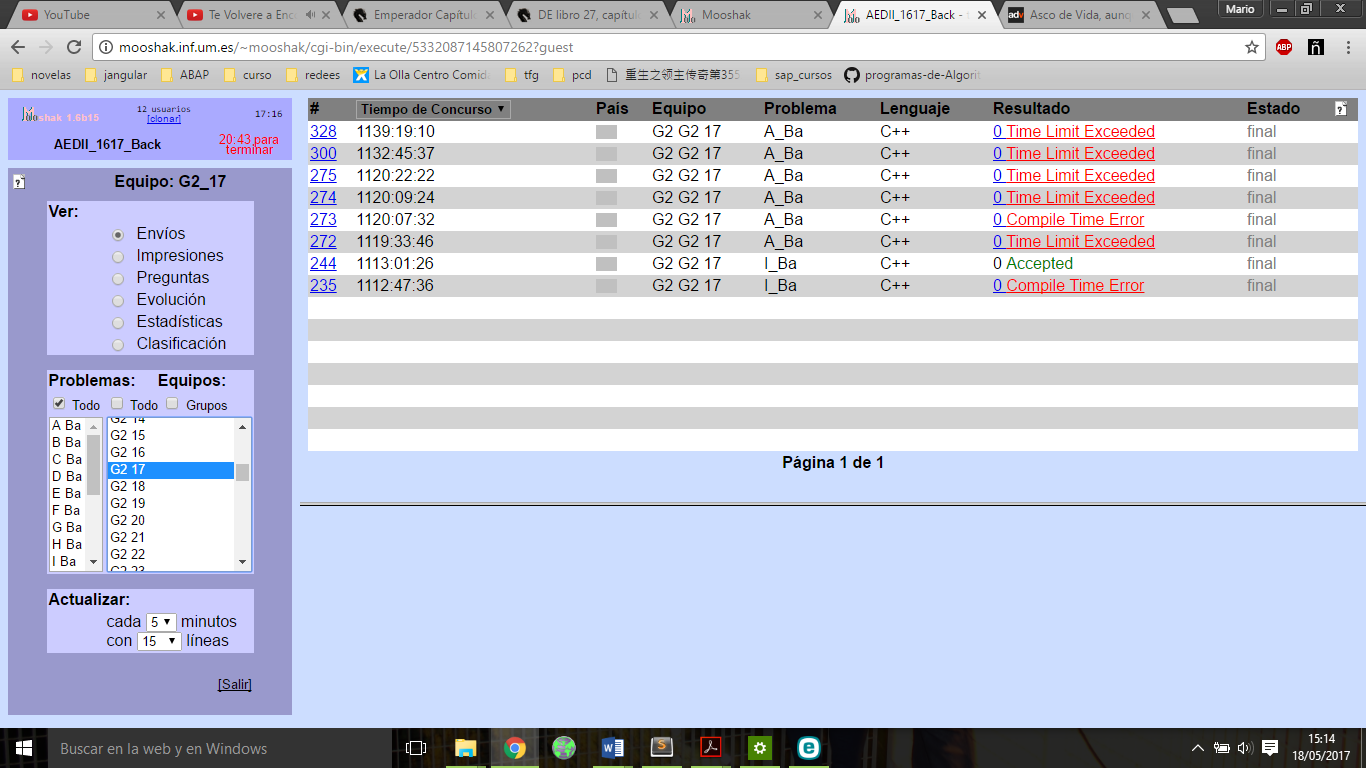
Estás trabajando de informático/a en una empresa que se dedica a reparar maquinaria pesada. Cada mecánico puede reparar ciertos tipos de avería. La reparación de una avería requiere de la asignación de un mecánico, y cada mecánico puede estar asignado cada día a un máximo de una avería. La empresa te pide que diseñes un programa que permita optimizar la asignación diaria de mecánicos a averías, de modo que se consiga reparar el máximo posible de averías cada día.

### El Problema

Tenemos M mecánicos y A averías por reparar. Una tabla bidimensional C, de elementos booleanos c (i, j), indica que el mecánico i tiene capacidad para reparar la averíaj.

Se debe realizar una asignación de mecánicos a averías que maximice el número de averías reparadas en el día, teniendo en cuenta que cada mecánico puede ser asignado a una única avería.

### Mooshak



El error es porque había declarado una variable que no utilizábamos.

# Avance rápido

## Pseudocódigo y explicación del algoritmo

En este apartado se pide mostrar el Pseudocódigo del algoritmo creado, pero primero queremos explicar el planteamiento que hemos realizado.

Primero escogemos un alumno, y buscamos un compañero a ese alumno entre todos los compañeros que se encuentran de pie. Para cada uno de ellos se calculará el beneficio que se obtiene al sentarlos juntos en un pupitre, y nos quedaremos con el compañero con el que mayor beneficio se ha obtenido. Se sentarán el alumno que buscaba compañero, y el compañero elegido en un pupitre, y ya no podrán ser candidatos a ser compañeros de otros alumnos. Es decir, una vez sentados no podrán ser reasignados.

Para la creación de nuestro algoritmo debemos explicar las funciones básicas que creemos pertinente:

**Funciones básicas**

**Variables**

alumnoSolo::: número de alumnos que no han sido sentados todavía.

numAlumnos::: número de alumnos.

Solución::: array donde se almacenan las parejas de alumnos sentados.

C::: array con los posibles candidatos de alumnos, se indica el estado de “cogido/ocupado”.

Compañero::: el compañero de cada alumno.

Amistad::: matriz contenedora de los valores de amistad entre los alumnos.

Trabajo::: matriz contenedora de los valores de trabajo entre los alumnos.

**Funciones**

Voraz::: es la función principal encargada de elegir alumnos, que se encuentren de pie, y asignarle un compañero a través de la función Factible.

Factible::: el objetivo de esta función es la de emparejar un alumno con otro intentando conseguir el mayor beneficio al sentarlos juntos. Comprueba que algún alumno esté de pie y si quedan varios, irá comprobando los beneficios de cada uno y al final seleccionará al de mayor orden.

**Pseudocódigo**

Voraz () {

Mientras ( AlumnosSolo != 0) {

Alumno = seleccionar ();

C:= C-{Alumno};

AlumnosSolo:= AlumnosSolo-1;

Factible (alumno);

}

}

Factible (alumno) {

Beneficio:=0;

MayorBeneficio:=0;

MejorCompanero = seleccionar ();

Si (No\_Alumnos\_De\_Pie) entonces {}

Si no {

Para cada alumno de pie {

Beneficio = CalcularBeneficio (alumno, alumnoDePie);

Si (Beneficio > MayorBeneficio) {

MejorCompanero:=alumnoDePie;

MayorBeneficio:=Beneficio;

}

}

C:=C-{alumnoDePie}

Solución:= S+ {alumno, alumnoDePie}

AlumnosSinPupitre:=AlumnosSinPupitre-1;

}

}

El punto de partida será con una solución vacía, S, a la cuál iremos añadiendo parejas de alumnos que están sentándose. Tendremos N/2 beneficio, pues tenemos 2 alumnos sentados por pupitre.

El emparejamiento empieza con el primer alumno. Se busca al compañero idóneo entre los que estén de pie y haciendo que se sienten juntos. Esta operación se realizara hasta sentar a todos los alumnos, pudiendo tener compañero o estar solo, ya que puede darse el caso en que el número de alumnos sea impar y entonces habrá un alumno sin compañero, al cual sentaremos solo.

## Programación del algoritmo

#include <stdlib.h> // Funcion exit

#include <iostream> // Variables cin y cout

#include <sys/time.h>

using namespace std;

#define MAX\_ALUMNOS 100

///////////////////////////////////////////////////////////

////////// VARIABLES GLOBALES ///////////////

///////////////////////////////////////////////////////////

struct pupitre{

int alumno;

int companero;

};

pupitre Solucion[MAX\_ALUMNOS/2];

int alumnosSinPupitre; //Entero que va a contener los alumnos que todavía no han sido sentados

int NAlumnos; //Entero que va a contener el numero de alumnos

bool C[MAX\_ALUMNOS]; //Array de booleanos que va a contener los alumnos candidatos(false si el alumno aun no ha sido sentado)

int Companero[MAX\_ALUMNOS]; //Array que va a contener el compañero de cada alumno(Inicializado a -1, porque 0 es un alumno)

int Amistad[MAX\_ALUMNOS][MAX\_ALUMNOS]; // Matriz que contiene los valores de amistad de los alumnos

int Trabajo[MAX\_ALUMNOS][MAX\_ALUMNOS]; // Matriz que contiene los valores de trabajo de los alumnos

int indiceSolucion;

int beneficioTotal;

///////////////////////////////////////////////////////////

/////////// FUNCIONES DEL PROGRAMA ///////////

///////////////////////////////////////////////////////////

/\*Funcion que va a inicializar las variables globales\*/

void resetearVariables(){

alumnosSinPupitre = 0;

indiceSolucion = 0;

beneficioTotal = 0;

for(int i = 0; i < MAX\_ALUMNOS/2; i++){

pupitre p;

p.alumno = 0;

p.companero = 0;

Solucion[i] = p;

}

for(int i = 0; i < MAX\_ALUMNOS; i++){

C[i] = 0;

Companero[i] = -1; //No lo ponemos a 0 porque el 0 es un compañero

}

for(int i = 0; i < MAX\_ALUMNOS; i++){

for(int j = 0; j < MAX\_ALUMNOS; j++){

Amistad[i][j] = 0;

Trabajo[i][j] = 0;

}

}

}

//Metodo que va a introducir los valores de las variables(Matrices, arrays,...)

void setVariables(int nAlumnos){

alumnosSinPupitre = NAlumnos;

//Matriz Amistad

for(int i = 0; i < nAlumnos; i++){

for(int j = 0; j < NAlumnos; j++){

if(j == i){ //La diagonal de las matrices es 0

Amistad[i][j] = 0;

}

else{

int amistad = 0;

cin >> amistad;

Amistad[i][j] = amistad;

}

}

}

//Matriz Trabajo

for(int i = 0; i < nAlumnos; i++){

for(int j = 0; j < NAlumnos; j++){

if(j == i){ //La diagonal de las matrices es 0

Trabajo[i][j] = 0;

}

else{

int trabajo = 0;

cin >> trabajo;

Trabajo[i][j] = trabajo;

}

}

}

}

/\*La función seleccionar se va a encargar de escoger, de entre los alumnos candidatos(C),

el primer alumno que encontremos que no esté sentado\*/

int seleccionar(){

int i = 0;

for(i = 0; i < NAlumnos-1 && C[i] == true; i++){}

if(i == NAlumnos-1 && C[i] == true) i = -1; //Si no quedan mas alumnos de pie, se devuelve -1

return i;

}

/\* La funcion factible se va a encargar de, dado un alumno, buscar todos los compañeros que no hayan sido

ya sentados, y calcular el beneficio que obtendría con cada alumno. Vamos a ir modificando el compañero

que asignaremos si vamos encontrando un compañero con el que se obtenga mayor beneficio.

\*/

void factible(int alumno){

int mejorCompanero;

int beneficio = 0;

int mayorBeneficio = 0;

int companero = seleccionar(); //Conseguimos el compañero por el que empezar a buscar

mejorCompanero = companero;

//Si no quedan mas alumnos de pie

if(companero == -1){

Companero[alumno] = -1;

}

else{

for(; companero < NAlumnos; companero++){

if(C[companero] == false){

beneficio = ( Amistad[alumno][companero] + Amistad[companero][alumno] )

\* ( Trabajo[alumno][companero] + Trabajo[companero][alumno] ) ;

if(beneficio >= mayorBeneficio){ //Si el beneficio que obtenemos con el compañero que estamos tratando es mayor que

//el beneficio que teniamos, actualizaremos el compañero y el beneficio maximo

mejorCompanero = companero;

mayorBeneficio = beneficio;

}

}

}

C[mejorCompanero] = true; //Sentamos al mejor compañero que hayamos encontrado

beneficioTotal += mayorBeneficio;

alumnosSinPupitre--;

}

pupitre p;

p.alumno = alumno;

p.companero = mejorCompanero;

Solucion[indiceSolucion] = p;

}

void printSolucion(){

//bool escogidos[NAlumnos];

int alumno;

int companero;

cout << beneficioTotal << endl;

int cant\_empa = 0;

cant\_empa = NAlumnos/2;

if(NAlumnos%2 != 0)

cant\_empa = cant\_empa+1;

//cout << NAlumnos << "-- > " << cant\_empa << endl;

for(int i = 0 ; i < cant\_empa; i++){

alumno = Solucion[i].alumno;

cout << alumno << " ";

if(Solucion[i].companero != -1){

companero = Solucion[i].companero;

cout << companero << " ";

}

//escogidos[alumno] = true;

//escogidos[companero] = true;

//cout << alumno << " " << companero << " ";

}

/\*alumno = Solucion[(NAlumnos/2)-1].alumno;

if(Solucion[(NAlumnos/2)-1].companero != -1){

companero = Solucion[(NAlumnos/2)-1].companero;

//escogidos[companero] = true;

cout << alumno << " " << companero;

}

else\*/

//cout << alumno;

//escogidos[alumno] = true;

/\*

if(NAlumnos%2 != 0){

for(int i = 0; i < NAlumnos; i++){

if(escogidos[i]!= true){

cout << " " << i << endl;

break;

}

}

}

else\*/

cout << endl;

}

void voraz(){

while(alumnosSinPupitre > 0 ){

int alumno = seleccionar(); //Se escoge, de entre C, el primer alumno que no esté sentado

C[alumno] = true; //El alumno se elimina del conjunto de candidatos

alumnosSinPupitre--;

factible(alumno); //Se busca un companero al alumno

indiceSolucion++;

}

printSolucion();

}

int main(void){

int numeroCasos;

cin >> numeroCasos;

while(numeroCasos >0){

resetearVariables();

cin >> NAlumnos;

setVariables(NAlumnos);

voraz();

numeroCasos--;

}

return 0;

}

## Estudio teórico del tiempo de ejecución del algoritmo

Tendremos un bucle en el algoritmo voraz que se repetirá con N alumnos/2 veces, es decir, tantas ocasiones como parejas de alumnos se formaran.

Se seleccionará al alumno mediante el método seleccionar (), el cuál buscará el primer alumno que esté de pie en el array e indicará qué alumnos están de pies o sentados. Por tanto el método seleccionar () Є O(n). Lo siguiente, es buscar un compañero al alumno seleccionado. Usando el método factible (), se buscará todos los compañeros que estén libres, la búsqueda será en base al alumno, así el método factible () Є O(n). En definitiva, el tiempo de ejecución del algoritmo es t(n) Є O (numAlumnos/2 \* (O (seleccionar ()) + O (factible ())). Si eliminamos las constantes nos queda que t(n) Є O(n).

## Estudio experimental del tiempo de ejecución

Hemos realizado distintas pruebas para ver el resultado de la relación de tamaño y tiempo:

|  |  |
| --- | --- |
| **Tamaño** | **Tiempo (mili segundos)** |
| 2 | 0,015 |
| 3 | 0,017 |
| 5 | 0,12 |
| 7 | 0,129 |
| 9 | 0,129 |
| 10 | 0,118 |
| 15 | 0,126 |
| 18 | 0,13 |

Se puede observar que el tiempo empleado por el algoritmo de avance rápido es bastante bueno, sobretodo, cuando el tamaño de los problemas es grande emplea muy poco tiempo en hallar una solución. En ocasiones el resultado no es el más óptimo pero teniendo en cuenta el tiempo empleado, se consigue que sea un buen algoritmo para hallar una solución.

Al realizar las pruebas, nos damos cuenta que este tipo de algoritmos será conveniente cuando el tamaño del problema sea muy grande.

## Comparación del estudio teórico y el experimental

Una vez realizado el estudio experimental, hemos comprobado que nos aproximamos bastante en el estudio teórico del tiempo de ejecución, ya que obteníamos un tiempo lineal, y como reflejan los resultados conseguidos experimentalmente, se puede constatar que el tiempo aumenta de manera más o menos lineal.

No se observa ningún gran salto en el tiempo respecto a los aumentos del tamaño del problema.

# Backtracking

## Pseudocódigo y explicación del algoritmo

Antes de comenzar con la explicación de los cambios que hemos introducido en el algoritmo base, vamos a recordar las ideas fundamentales de Backtracking.

Backtracking realiza una búsqueda exhaustiva y sistemática en el espacio de soluciones. Prueba todas las combinaciones. Por ello, suele resultar muy ineficiente.

En nuestro problema el árbol implícito será un árbol permutacional en el que cada nivel se decide que mecánico se asigna a cada avería, es decir, cada nivel del árbol representara a una avería.

La solución será un vector *S* con índices de 1 a m (uno por cada avería), y los valores de 1 a n que serán los mecánicos que se asignan.

El valor 0 indicara que la avería no ha sido asignada para su reparación.

La condición de fin será que volvamos al nodo raíz o que se hayan asignado todas las averías, ya que lo que buscamos es buscar una solución con la máxima cantidad de averías asignadas.

Se recorrerá todo el árbol comprobando en cada nodo si hemos llegado a un nodo hoja, es decir, si en la solución se han evaluado todas las averías.

Para mantener la solución más óptima mientras se recorre todo el árbol hemos utilizado las siguientes variables:

* ***Voa***: es la cantidad de mecánicos que se han sido asignados.
* ***Soa***: contiene los mecánicos seleccionados para reparar la avería

Además hemos utilizado algunas variables extras para mantener la información en cada nodo:

* ***Mec\_asig***: es un vector de tamaño n (mecánicos) que permite mantener la información que mecánicos han sido asignados, ya que una de las restricciones del problema es que un mecánico no pueda estar asignada a más de una avería.
* ***Av\_mec***: cantidad actual de mecánicos asignados en un nodo i.

El algoritmo es similar a un esquema de maximización en la asignación de tareas.

## Pseudocódigo

Vamos a explicar el problema mediante Pseudocódigo:

backtracking(vector<Mecanico> mecanicos, **int** nAverias, **int** nMec){

vector<**int**> S, mec\_asig;

S.assign(nAverias, 0);

mec\_asig.assign(nMec, 0);

**int** nivelMax = nAverias;

//nivel por el que va el algoritmo.

**int** nivel=1;

//estas variables son la que contienen la solucion optima.

**int** voa = 0;

vector<**int**> soa;

soa.assign(nAverias, 0);

**int** av\_asig=0;

**while** (nivel > 0 && voa != nAverias && voa != nMec) {

generar(nivel, S, mecanicos, mec\_asig);

av\_asig = valor(S);

//se comprueba si la varaible solucion es una solucion y si es mas optima que la que tenemos

**if** (solucion(nivel, nivelMax) && (av\_asig > voa)){

voa = av\_asig;

soa = S;

}

//comprobamos si podemos llegar a una solucion a partir del nodo actual

**else** **if** (criterio(nivel,nivelMax, voa)){

nivel++;

}

**else** {

//se comprueba si existe mas hermanos.

**while**(!masHermanos(nivel, S, mecanicos, mec\_asig) && nivel > 0 )

retroceder(nivel, S, mec\_asig);

}

}

cout << voa << endl;

**for** (**int** i=0; i < nAverias; i++){

cout << soa.at(i);

**if**(i != nAverias-1)

cout << " ";

}

cout << endl;

}

## Programación del algoritmo

## Estudio teórico del tiempo de ejecución del algoritmo

## Estudio experimental del tiempo de ejecución

## Comparación del estudio teórico y el experimental

/\*

\* Función Generar

\* Función que genera un nodo con una posible solución.

\*/

**void** generar(**int** nivel, vector<**int**> &s, vector<Mecanico> mecanicos, vector<**int**> &asignaciones ){

**int** valor, mecanico;

valor= s[nivel-1];

mecanico = seleccionMecanico(nivel-1, valor, mecanicos, asignaciones);

//si ningun mecanico tiene la capacidad para reparar la averia.

**if**(mecanico > 0){

s.at(nivel-1) = mecanico;

asignaciones.at(mecanico-1) = 1;

}**else**

s.at(nivel-1) = mecanico;

**if** (valor > 0 )

asignaciones.at(valor-1) = 0;

}