Práctica 1: Backtracking y Branch and bound

Rubén Dugo Martín
Profesor: Rafael Alcalá
Teoría de Algoritmos
Curso 2009-2010
Universidad de Granada

1. Introducción

Sobre la implementación de la práctica no hay nada que resaltar especialmente, las fuentes están totalmente comentada, tanto el código como las funciones (mediante *Doxygen*).

Cabe destacar el uso de una macro especial en los ficheros, la macro *__VERBOSE__* que declarada mediante *#define* sin valor hará que el compilador elija el código de forma que las funciones devuelvan los valores de nodos expandidos, nodos podados y tamaño máximo de la cola de nodos vivos. De esta forma se compila correctamente para después enlazarlo con un ejecutable que recoja esos datos.

2. Backtracking

Utilizando el programa *mainBK* (listado en el apéndice) obtenemos la siguiente salida:

```
n=11:  [( 1 2 3 4 5 6 7 8 9 9 10 ),tiempo=11] 148685 1230655 0.88569
n=12:  [( 8 9 10 4 5 6 7 6 1 2 3 4 ),tiempo=19] 1414884 11356376 7.6217
n=13:  [( 9 10 8 2 3 4 5 6 7 8 9 7 1 ),tiempo=58] 10027353 77719167 52.483
n=14:  [( 5 6 7 7 8 9 10 10 4 5 1 2 3 4 ),tiempo=62] 61170687 456824603
314.24231
n=15:  [( 5 2 3 3 4 5 6 7 8 9 10 9 10 6 1 ),tiempo=67] 338327691 2431813269
1717.13
```

Sólo hasta *n=15* ya que casos mayores podría tardar incluso días.

Los tres últimos valores son nodos expandidos, nodos podados y tiempo total invertido.

La solución que muestra (entre corchetes) es sólo la del último caso, sin embargo los demás datos son las medias de 20 ejecuciones.

Calculo el tiempo de ejecución por nodo expandido para cada tamaño.

Sabiendo que $t_{nodo}(n) = \frac{t(n)}{N}$

```
11 0.000005956
12 0.000005386
13 0.000005233
14 0.000005137
15 0.000005075
```

Introduzco los datos en *xmgrace* y busco la mejor aproximación:

$$f_t(n) = 2.84498 \cdot 10^{-15} n^3 - 1.11504 \cdot 10^{-13} n^2 - 2.01099 \cdot 10^{-6} n + 7.97169 \cdot 10^{-6}$$

El mismo proceso hago para el número de nodos:

$$f_N(n) = 0.999993 n^4 + 1372 \mathrm{n}^3 - 31197.6 \mathrm{n}^2 + 259969 - 747975$$

De modo que podemos obtener la función del tiempo de la forma: $t(n) = f_N(n) \cdot f_t(n)$

3. Branch & Bound

Utilizando el programa *mainBB* (listado en el apéndice) obtenemos la siguiente salida:

```
n=11:  [( 4 3 7 9 10 1 8 6 2 5 1 ),tiempo=29] 905584 4121596 33515 4.436
n=12:  [( 3 10 8 9 5 7 1 4 3 2 2 6 ),tiempo=30] 1767032 5806408 33914 2.411
n=13:  [( 7 9 2 10 1 3 6 8 4 9 10 5 1 ),tiempo=58] 2274962 6843728 33914
1.378
n=14:  [( 1 4 7 1 3 5 2 8 9 6 3 10 1 2 ),tiempo=62] 4776537 12211293 88138
8.31
n=15:  [( 3 4 3 5 10 7 1 3 1 2 9 2 6 4 8 ),tiempo=61] 11567975 26870512
385172 24.483
n=16:  [( 3 5 2 6 10 7 1 1 8 2 4 4 3 3 9 4 ),tiempo=62] 15240868 34844051
385172 13.295
n=17:  [( 2 5 1 2 3 7 3 1 4 3 1 10 9 6 4 8 2 ),tiempo=82] 31276236 68040873
1390276 58.544
```

Este llega a casos mayores que el *Backtracking* ya que es más rápido.

Los cuatro últimos valores son nodos expandidos, nodos podados, tamaño máximo de la lista de nodos vivos y tiempo total invertido.

La solución que muestra (entre corchetes) es sólo la del último caso, sin embargo los demás datos son las medias de 20 ejecuciones.

Calculo el tiempo de ejecución por nodo expandido para cada tamaño.

Sabiendo que $t_{nodo}(n)=\frac{t(n)}{N}$

```
11 0.0000048984964
12 0.0000013644348
13 0.0000006057244
14 0.0000017397541
15 0.0000021164465
16 0.0000008723256
17 0.0000018718365
```

Introduzco los datos en *xmgrace* y busco la mejor aproximación:

$$f_t(n)=5.39655\cdot 10^{-14}n^4-1.12372\cdot 10^{-7}n^3-4.94249\cdot 10^{-6}n^2-7.18345\cdot 10^{-5}n+0.000346334$$

El mismo proceso hago para el número de nodos:

$$f_N(n) = -0.00419397\mathrm{n}^5 + 1932.18\mathrm{n}^4 - 19636.6\mathrm{n}^3 - 252952\mathrm{n}^2 + 1.34874 \cdot 10^{6x} + 1.52099 \cdot 10^7$$

De modo que podemos obtener la función del tiempo de la forma: $t(n) = f_N(n) \cdot f_t(n)$

4. Apéndice

Fichero mainBK.cpp

```
#include <iostream>
#include <vector>
#include <cstdlib>

#define VECES 20

using namespace std;

void bombillas_BK (const std::vector<int> T,std::vector<int> &A, int &tiempo,
register unsigned int &NEXP, register unsigned int &NPODAS);

inline void init(vector<int> &V)
{
      for (int i=0;i<V.size();i++)
            V[i] = 1+static_cast<int>( (100.0*rand()/(RAND_MAX+1.0)));
}

int main()
{

      srand(time(NULL));
      vector<int> A;
      vector<vector<int> > T(VECES);
      int tiempo=0;
      register int j;
      clock_t antes, despues;

      register unsigned int NEXP=0,NPODAS=0; // Contadores de nodos expandidos y
nodos podados resp
      int TEXP=0,TPODAS=0;

      for (int i=11;i<20;i++)
      {
            for (j=0;j<VECES;j++)
            {
                  T[j].resize(i);
                  init(T[j]);
            }
```

```
            A.resize(i);

            cout << "n=" << i <<": ";

            antes = clock();
            for ( j =0 ; j<VECES; j++)
            {
                  bombillas_BK(T[j],A,tiempo,NEXP,NPODAS);
                  TEXP+=NEXP; TPODAS+=NPODAS;
            }
            despues = clock();

            cout << " [( ";
            for (j=0;j<i;j++)
                  cout<<T[VECES-1][j]<<'-'<<A[j]<<' ';
            cout << "),tiempo=" <<tiempo<<"] "<<TEXP/VECES<<' '<<TPODAS/VECES<<'
'<<( ( double ) ( despues-antes) / (double)(CLOCKS_PER_SEC*VECES) )<<endl;

      }
}
```

Fichero bombillas_BK.cpp

```
#include <vector>


/**

@brief Macro para compilar el programa para un determinado número de focos.


Por defecto 10.

*/

#define NFOCOS 10


/**

@brief Macro para compilar el programa en modo "verbose".


Compila al programa para que devuelva el número de podas y expansiones
realizadas.

Por defecto desactivada.

*/

#define __VERBOSE__


#ifdef __VERBOSE__

#include <iostream>

#endif




#ifdef __VERBOSE__

void bombillas_BK (const std::vector<int> T,std::vector<int> &A, int &tiempo,
register unsigned int &NEXP, register unsigned int &NPODAS);

void BK(const std::vector<int> &T,std::vector<int> X,std::vector<int>
&A,std::vector<int> &F,int &tiempo, int k, register unsigned int &NEXP, register
unsigned int &NPODAS);

#else

void bombillas_BK (const std::vector<int> T,std::vector<int> &A, int &tiempo);

void BK(const std::vector<int> &T,std::vector<int> X,std::vector<int>
&A,std::vector<int> &F,int &tiempo, int k);

#endif


inline int Min(const std::vector<int> &V);

inline bool Factible(const std::vector<int> &X,const int &flibres, const int
&k);


#ifdef __VERBOSE__

void bombillas_BK (const std::vector<int> T,std::vector<int> &A, int &tiempo,
register unsigned int &NEXP, register unsigned int &NPODAS)

#else
```

```
void bombillas_BK (const std::vector<int> T, std::vector<int> &A, int &tiempo)
#endif
{
	int k = 0;
	tiempo = 0; // Objetivo a maximizar

	std::vector<int> X(T.size(),0), F(NFOCOS,0);


	#ifdef __VERBOSE__
	BK(T,X,A,F,tiempo,k,NEXP,NPODAS);
	#else
	BK(T,X,A,F,tiempo,k);
	#endif
}

/**
@brief Función recursiva de la implementación backtracking.
@param T Vector con los tiempos de los focos.
@param X Vector de la solución parcial.
@param A Vector con la solución final (óptima). ES MODIFICADO
@param F Vector con los tiempos de los focos acumulados. ES MODIFICADO
@param tiempo Tiempo mínimo del vector de soluciones. ES MODIFICADO
@param k Fase (en profundidad) por la que va el algoritmo, inicialmente 0.
@param NEXP Número de nodos total que se expanden en la ejecución del algoritmo.
ES MODIFICADO
@param NPODAS Número de podas total que se realizan en la ejecución del
algoritmo. ES MODIFICADO

Los dos últimos parámetros (NEXP y NPODAS) sólo serán necesarios cuando se
active la macro __VERBOSE__.
*/
#ifdef __VERBOSE__
void BK(const std::vector<int> &T,std::vector<int> X,std::vector<int>
&A,std::vector<int> &F,int &tiempo,int k, register unsigned int &NEXP, register
unsigned int &NPODAS)
#else
void BK(const std::vector<int> &T,std::vector<int> X,std::vector<int>
&A,std::vector<int> &F,int &tiempo,int k)
#endif
{
	if(k>=X.size())
	{
		int min = Min(F);
```

```
	// Si es esta solucion mejor que la que tengo ya
	if(min>tiempo)
	{
		// La almaceno, funcion Actualizar()
		tiempo = min;
		A = X;
	}
}
else
{
	for(register int i=1;i<=NFOCOS;i++)
	{
		// Cuento los focos libres para la funcion Factible()
		int flibres = 0;
		for (register int j=0;j<NFOCOS;j++)
			if (F[j] == 0) flibres++;

		if (F[i-1] == 0) flibres--;

		X[k] = i;
		if(Factible(X,flibres,k))
		{
			#ifdef __VERBOSE__
			NEXP++;
			#endif

			// Sumo a F el tiempo de la bombilla k
			F[i-1] += T[k];

			// Recursividad!
			#ifdef __VERBOSE__
			BK(T,X,A,F,tiempo,k+1, NEXP, NPODAS);
			#else
			BK(T,X,A,F,tiempo,k+1);
			#endif

			// Resto a F el tiempo de la bombilla k
			F[i-1] -= T[k];

		}
		#ifdef __VERBOSE__
		else NPODAS++;
```

```
                #endif
            }
      }
}

/**
@brief Obtiene el mínimo de un vector.
@param V Vector.
@return Mínimo del vector.
*/
inline int Min(const std::vector<int> &V)
{
      int minimo = V[0];

      for(unsigned int i=1;i<V.size();i++)
            if(V[i]<minimo) minimo = V[i];


      return minimo;
}

/**
@brief Comprueba si una solución parcial cumple las restricciones implícitas.
@param X Vector de la solución parcial.
@param flibres Focos libres en dicha solución.
@param k Fase (profundidad en el árbol de estados) por la que iba la solución.
*/
inline bool Factible(const std::vector<int> &X,const int &flibres, const int &k)
{
      register unsigned int blibres = 0, i = 0;

      // Cuento las bombillas libres
      for (;i<X.size();i++)
            if (X[i]==0) blibres++;

      // Restriccion (1); deben quedar igual o mas bombillas sin asignar que
focos libres
      if (blibres < flibres);
      // Restriccion (2); no se podra colocar una bombilla en un foco > foco
anterior +1
      // esto evita soluciones repetidas
      else if (k>0 && X[k]>X[k-1]+1);
      else return true;
```

```
        return false;
    }
```

Fichero mainBB.cpp

```
#include <iostream>
#include <vector>
#include <cstdlib>

#define VECES 20

using namespace std;

void bombillas_BB (const std::vector<int> T,std::vector<int> &A, int &tiempo,
register unsigned int &NEXP, register unsigned int &NPODAS, register unsigned
int &MAXNV);

inline void init(vector<int> &V)
{
      for (int i=0;i<V.size();i++)
            V[i] = 1+static_cast<int>( (100.0*rand()/(RAND_MAX+1.0)));
}



int main()
{

      srand(time(NULL));
      vector<int> A;
      vector<vector<int> > T(VECES);
      int tiempo=0;
      register int j;
      clock_t antes, despues;

      register unsigned int NEXP=0,NPODAS=0,MAXNV=0; // Contadores de nodos
expandidos y nodos podados resp

      int TEXP=0,TPODAS=0,TMAXNV=0;

      for (int i=11;i<20;i++)
      {
            for (j=0;j<VECES;j++)
            {
                  T[j].resize(i);
                  init(T[j]);
            }
```

```
            A.resize(i);

            cout << "n=" << i <<": ";

            antes = clock();
            for ( j =0 ; j<VECES; j++)
            {
                  bombillas_BB(T[j],A,tiempo,NEXP,NPODAS,MAXNV);
                  TEXP+=NEXP; TPODAS+=NPODAS; TMAXNV+=MAXNV;
            }
            despues = clock();

            cout << " [( ";
            for (j=0;j<i;j++)
                  cout<<A[j]<<' ';
            cout << "),tiempo=" <<tiempo<<"] "<<TEXP/VECES<<' '<<TPODAS/VECES<<'
'<<TMAXNV/VECES<<' '<<( ( double ) ( despues-antes) / (double)
(CLOCKS_PER_SEC*VECES) )<<endl;


      }
}
```

Fichero bombillas_BB.cpp

```
#include <vector>
#include <queue>

/**
@brief Macro para compilar el programa para un determinado número de focos.

Por defecto 10.
*/
#define NFOCOS 10

/**
@brief Macro para compilar el programa en modo "verbose".

Compila al programa para que devuelva el número de podas, expansiones realizadas
y el tamaño mayor alcanzado en la lista de nodos vivos.
Por defecto desactivada.
*/
#define __VERBOSE__



#ifdef __VERBOSE__
#include <iostream>
#endif

/**
@brief Estructura necesaria para almacenar un nodo en la lista de nodos vivos.
*/
struct Nodo
{
      Nodo() { };
      Nodo(std::vector<int> x, std::vector<int> f, double c, double m, int q)
{ X=x;F=f;CLocal=c;M=m;k=q; }
      std::vector<int> X,F;
      double CLocal, M;
      int k;
};

/**
@brief Clase para que priority_queue pueda ordenar los nodos.
*/
```

```
class ComparadorNodo
{
	public:
		bool operator()(Nodo a, Nodo b)
		{
			return a.M < b.M;
		}
};


inline bool Factible(const std::vector<int> &X,const int &flibres, const int &k);
inline double CotaSuperior(const std::vector<int> &T, const std::vector<int> &X, std::vector<int> F);
inline double CotaInferior(const std::vector<int> &T, std::vector<int> F, const int &k);
void bombillas_BB_qs (std::vector<int> &S, std::vector<int> &P,const unsigned int &ini, const unsigned int &fin);
void bombillas_BB_qs (std::vector<int> &S, const unsigned int &ini, const unsigned int &fin);
inline int Min(const std::vector<int> &F);



#ifdef __VERBOSE__
void bombillas_BB (const std::vector<int> T,std::vector<int> &A, int &tiempo, register unsigned int &NEXP, register unsigned int &NPODAS, register unsigned int &MAXNV)
#else
void bombillas_BB (const std::vector<int> T, std::vector<int> &A, int &tiempo)
#endif
{
	Nodo actual; // Nodo que estamos procesando actualmente
	register int i;
	int miny;
	double CGlobal, CIy, CSy; // Cota global, cota inferior y cota superior para un nodo y
	unsigned int n = T.size();
	tiempo = 0;

	// Cola con prioridad para la lista de nodos vivmos
	std::priority_queue<Nodo, std::vector<Nodo>, ComparadorNodo> vivos;



	std::vector<int> TOrd(T), TPos(T.size()), X(n,0),F(NFOCOS,0);

	for (i=0;i<T.size();i++)
```

```
            TPos[i] = i;

      // Ordeno el vector de bombillas que será necesario para la función
CotaInferior
      bombillas_BB_qs(TOrd, TPos, 0, n-1);

      // Convierto el orden ascendente en descendente (invierto)
      for (i=0;i<n/2;i++)
      {
            std::swap(TOrd[i],TOrd[n-i-1]);
            std::swap(TPos[i],TPos[n-i-1]);
      }

      CGlobal = CotaInferior(TOrd,F,-1);

      // Relleno el nodo raiz y lo meto en la cola
      actual.F = F;
      actual.X = X;
      actual.k = -1;
      actual.CLocal = CotaSuperior(TOrd,X,F);
      actual.M = (actual.CLocal+CGlobal)/2.0;
      vivos.push(actual);

      // Bucle principal BB
      while(!vivos.empty())
      {
            #ifdef __VERBOSE__
            MAXNV = (vivos.size()>MAXNV)?vivos.size():MAXNV;
            #endif

            // Cojo el nodo mejor estimado
            actual = vivos.top();
            vivos.pop();

            // Si su cota es mejor o igual que la cota global actual
            // lo proceso, en otro caso lo descarto
            if (actual.CLocal >= CGlobal)
            {
                  // Genero los hijos, uno por foco
                  for (i=1;i<=NFOCOS;i++)
                  {
                        // Cojo los valores que me interesan
```

```
                        X = actual.X;
                        X[actual.k+1] = i;
                        F = actual.F;
                        F[i-1] += TOrd[actual.k+1];

                        // Cuento los focos libres para la funcion Factible()
                        // igual que en BK
                        int flibres = 0;
                        for (register int j=0;j<NFOCOS;j++)
                              if (F[j] == 0) flibres++;

                        if (Factible(X, flibres, actual.k +1))
                        {
                              #ifdef __VERBOSE__
                              NEXP++;
                              #endif
                              if (actual.k+2 >= n) // Se trata de un nodo hoja
                              {
                                    // Si la hoja alcanzada es mejor que la
solucion actual
                                    if ((miny=Min(F))>tiempo)
                                    {
                                          // La almaceno
                                          tiempo = miny;
                                          A = X;
                                          // y actualizo la cota global
                                          CGlobal = CotaInferior(TOrd,F,actual.k
+1);
                                    }
                              }
                              else
                              // Si no es hoja compruebo su cota
                              if ((CSy = CotaSuperior(TOrd,X,F)) >= CGlobal)
                              {
                                    CIy = CotaInferior(TOrd,F,actual.k +1);
                                    // Lo meto en la lista de nodos vivos con
sus valores
                                    vivos.push(Nodo(X, F, CSy,
(CSy+CGlobal)/2.0, actual.k +1));
                                    // Si es mas prometedor que la cota global
actual
                                    if (CIy > CGlobal)
                                    {
                                          // Lo tomo como la solucion actual
```

```
                                    CGlobal = CIy;
                                    tiempo = Min(F);
                                    A = X;
                              }
                        }
                  }
                  #ifdef __VERBOSE__
                  else
                  NPODAS++;
                  #endif
            }
      }
      #ifdef __VERBOSE__
      else
      NPODAS++;
      #endif
}

// Reordenados la solucion
X = A;
for (i=0;i<n;i++)
{
      A[TPos[i]] = X[i];
}
}

/**
@brief Obtiene la cota superior según lo definido en el enunciado de la práctica
@param T Vector con los tiempos de las bombillas.
@param X Vector solución parcial.
@param F Vector con las duraciones de los focos acumulados para la solución parcial X.
@return La cota superior.
*/
inline double CotaSuperior(const std::vector<int> &T, const std::vector<int> &X, std::vector<int> F)
{
      register unsigned int i = 0, resto = 0;
      double obj;
      int aux;

      // Calculo el tiempo de los focos que me quedan
      for (;i<T.size();i++)
```

```
            if (X[i]==0) resto += T[i];

      // Ordenamos F de forma ascendente
      bombillas_BB_qs(F,0,NFOCOS-1);

      // Este es el algoritmo descrito en el enunciado de la práctica
      obj = F[0];
      for (i=0;i<NFOCOS-1 && resto>0;i++)
      {
            aux = (i+1) * (F[i+1]-obj);
            aux = (resto<aux)?resto:aux;
            resto -= aux;
            obj+= static_cast<double>(aux)/static_cast<double>(i+1);
      }

      if (resto>0) obj += static_cast<double>(resto)/10.0;

      return obj;
}

/**
@brief Obtiene la cota superior según lo definido en el enunciado de la práctica
@param T Vector con los tiempos de las bombillas.
@param k Bombilla que se está procesando actualmente.
@return La cota inferior.
*/
inline double CotaInferior(const std::vector<int> &T, std::vector<int> F, const
int &k)
{
      register unsigned int i = 0;

      // Ordenamos F de forma ascendente
      bombillas_BB_qs(F,0,NFOCOS-1);

      // Comienza el algoritmo Greedy partiendo desde k
      for (i=k+1;i<T.size();i++)
      {
            F[0] += T[i];

            // Volvemos a ordenar
            bombillas_BB_qs(F,0,NFOCOS-1);
      }
```

```
	return static_cast<double>(F[0]);
}

/**
@brief Obtiene el mínimo para un vector.
@param F Vector para obtener el mínimo
@return El mínimo del vector.
*/
inline int Min(const std::vector<int> &F)
{
	register int i = 0, minimo = F[0];

	for(;i<F.size();i++)
		if(F[i]<minimo) minimo = F[i];

	return minimo;
}



/**
@brief Ordena mediante el algoritmo quicksort en forma creciente
@param S Vector a ordenar. ES MODIFICADO
@param P Vector de posiciones (explicación abajo). ES MODIFICADO
@param ini Marca el comienzo del subconjunto a ordenar (inclusive). NO ES MODIFICADO
@param fin Marca el final del subconjunto a ordenar (inclusive). NO ES MODIFICADO

Esta función ordena de forma creciente el vector S y también modifica el vector P
según el mismo criterio, de esta forma se puede reconstruir el vector original S.

NOTA: Esta función es la función utilizada por mi para ordenar en la práctica 1 modificada
según las necesidades de esta.
*/
void bombillas_BB_qs (std::vector<int> &S, std::vector<int> &P, const unsigned int &ini, const unsigned int &fin)
{
    /*** Casos particulares ***/

    // Conjunto con un único elemento o vacío
```

```
    if (fin-ini+1<=1) return;

    // Conjunto con dos elementos
    if (fin-ini+1==2)
    {
        if (S[ini]>S[fin])
        {
            std::swap(S[ini],S[fin]);
            std::swap(P[ini],P[fin]);
        }
        return;
    }

    unsigned int der=fin,izq=ini+1,posmedia,i;
    float pivote=S[ini];

    // Realizo el intercambio de elementos
    while (izq < der)
    {
        while (izq<der && S[izq] <= pivote) izq++;
        while (izq<der && S[der]  > pivote) der--;

        std::swap(S[izq],S[der]);
        std::swap(P[izq],P[der]);
    }

    // Elijo la posicion media
    posmedia = (pivote>S[izq])?izq:izq-1;

    if (ini!=posmedia)
    {
        std::swap(S[ini], S[posmedia]);
        std::swap(P[ini], P[posmedia]);
    }

    bombillas_BB_qs(S, P, ini, posmedia-1);
    bombillas_BB_qs(S, P, posmedia+1, fin);
}

/**
@brief Ordena mediante el algoritmo quicksort en forma creciente
@param S Vector a ordenar. ES MODIFICADO
```

```
@param ini Marca el comienzo del subconjunto a ordenar (inclusive). NO ES MODIFICADO
@param fin Marca el final del subconjunto a ordenar (inclusive). NO ES MODIFICADO


NOTA: Esta función es la función utilizada por mi para ordenar en la práctica 1 modificada
según las necesidades de esta.
*/
void bombillas_BB_qs (std::vector<int> &S, const unsigned int &ini, const unsigned int &fin)
{
    /*** Casos particulares ***/

    // Conjunto con un único elemento o vacío
    if (fin-ini+1<=1) return;

    // Conjunto con dos elementos
    if (fin-ini+1==2)
    {
        if (S[ini]>S[fin])
        {
            std::swap(S[ini],S[fin]);
        }
        return;
    }

    unsigned int der=fin,izq=ini+1,posmedia,i;
    float pivote=S[ini];

    // Realizo el intercambio de elementos
    while (izq < der)
    {
        while (izq<der && S[izq] <= pivote) izq++;
        while (izq<der && S[der]  > pivote) der--;

        std::swap(S[izq],S[der]);
    }

    // Elijo la posicion media
    posmedia = (pivote>S[izq])?izq:izq-1;

    if (ini!=posmedia)
```

```
    {
        std::swap(S[ini], S[posmedia]);
    }

    bombillas_BB_qs(S, ini, posmedia-1);
    bombillas_BB_qs(S, posmedia+1, fin);
}

/**
@brief Comprueba si una solución parcial cumple las restricciones implícitas.
@param X Vector de la solución parcial.
@param flibres Focos libres en dicha solución.
@param k Fase (profundidad en el árbol de estados) por la que iba la solución.
*/
inline bool Factible(const std::vector<int> &X,const int &flibres, const int &k)
{
      register unsigned int blibres = 0, i = 0;

      // Cuento las bombillas libres
      for (;i<X.size();i++)
            if (X[i]==0) blibres++;

      // Restriccion (1); deben quedar igual o mas bombillas sin asignar que
focos libres
      if (blibres < flibres);
      // Restriccion (2); no se podra colocar una bombilla en un foco > foco
anterior +1
      // esto evita soluciones repetidas
      else if (k>0 && X[k]>X[k-1]+1);
      else return true;

      return false;
}
```