Ejercicios Algorítmica

Javier Gómez Luzón
2º C

17.- Tenemos un vector de *N* elementos, y pretendemos comprobar si existe algún elemento en él que se repita al menos *N/2* veces. Diseñe un algoritmo que realice dicha comprobación, y devuelva, en su caso, cuál es el elemento repetido. Si existen dos elementos, devolverá cualquiera de ellos. El orden de complejidad del algoritmo, en el caso medio, no debe ser superior al lineal si el elemento existe.

Iremos dividiendo el vector en mitades iguales hasta llegar a un caso base (un sub vector de un solo elemento) e ir sumando las ocurrencias partiendo de 0 con el vector completo. Devolveremos el número si es distinto de null (pondremos este valor, ya que es un valor distinto de cualquier numero que se puede incluir en un vector de enteros) y si llega a N/2 devolveremos ese número.

```
int repetidos(int vI, vF, vector<int> & v, int n){
	if(vI-vF==0){
		int repetidos=0;
		for(int i=0;i<n;i++){
			if(i!=vI && v[i]==v[vI]) repetidos++;
		}
		if(repetidos>=n/2) return v[vI];
		else return null;
	}
	int mitad=(vI+vF)/2;
	int resultado=repetidos(vI,mitad,v,n);
	if(repetidos!=null) return repetidos;
	else return repetidos(mitad+1, vF,vn);
}
```

14.− Sean X e Y dos vectores de enteros sin elementos repetidos de dimensión *N*. Se sabe que X está ordenado creciente mente y que Y lo está decreciente mente, al menos uno en sentido estricto. Construya una función recursiva que decida en tiempo logarítmico si hay un índice i tal que

X[i] = Y[i]

Iremos dividiendo el vector en sub vectores más pequeños, hasta llegar al caso base (un vector de un solo elemento) y haremos la comprobación. Si la encontramos devolveremos true instantáneamente, en caso contrario seguiremos buscando hasta haber comprobado todos los elementos.

```
bool elementosDePosiciones(int vI, int vF, vector<int> & x, vector<int> & y){
        if(vI-vF==0){
                if(x[vI]==y[vI]) return true;
                else return false;
        }
        int mitad=(vI+vF)/2;
        bool solucion=elementosDePosiciones(mitad,vF,x,y);
        if(devolver) return true;
        return elementosPosicion(vI,mitad-1,x,y);
}
```

11.− Disponemos de un vector A de *N* elementos enteros ordenados en sentido creciente estricto, i.e., no hay elementos repetidos. Se pretende encontrar una posición i, tal que A[i] = i. Diseñe un algoritmo "*Divide y Vencerás*" que devuelva dicha posición o retorne el valor 0 cuando no exista ninguna.

Iremos dividiendo el vector por la mitad en sub vectores, a continuación iremos haciendo llamadas recursivas hasta llegar al caso base (un sub vector de tamaño 1). Cuando estemos en un caso base se hará la comprobación para ver si el valor de ese indice es igual al valor del vector en ese indice. Si se cumple se devuelve true, si no se devuelve false y se continuará buscando.

```
int posicionIgualAElemento(int vI, int vF, vector<int> & v){
        if(vI-vF==0){
                if(v[vI]==vI) return vI;
                else return 0;
        }
        int mitad=(vI+vF)/2;
        int solucion=posicionIgualAElemento(mitad,vF,v);
        if(solucion>0) return solucion;
        return posicionIgualAElemento(vI,mitad-1,v);
}
```

5-Resuelva el problema de la mochila versión 0/1 usando Backtracking

El espacio de búsqueda lo vamos a representar con un árbol binario. En el que cada nivel representará la decisión de coger un objeto o no cogerlo. La solución se representa con un vector s, donde s[i] es la decisión tomada para el objeto i.
Donde Objeto es un struct donde hay un float de peso y beneficio.
Factible nos dirá si el objeto que estamos dudado si elegir o no cabe en la mochila.
Beneficio nos devolverá el beneficio total que llevamos hasta ahora.

```
vector<int> mochilaBT(const vector<Objeto> & objs, float m){
	vector<int> s(objs.size()-1), sOptimo;
	int nivel=0;
	while(nivel>=0){
		s[nivel]++;
		if(s[nivel]==2){
			nivel--;
			s[nivel]==-1;
		}
		else{
			if(factible(s,objs,m)){
				if(nivel==objs.size()-1){
					if(beneficio(s)>beneficio(sOptimo)) sOptimo=s;
				}
				else nivel++;
			}
		}
	}
	return s;
}
```

26.- Para *Forrest Gump* la vida era como una caja de bombones. Cada caja tenía bombones de varias clases. *Forrest* frecuentaba una tienda que tenía muchas cajas distintas, y *Forrest* siempre deseó tener bombones de todas las clases, pero su paga no le alcanzaba. Por ello, tenía que comprar el menor número de cajas pues todas las cajas tenían el mismo precio. Diseñe una heurística voraz que le diga a *Forrest* qué cajas debe comprar y cuánto le costará comprarlas.

Candidatos =C=Las cajas de bombones que aun están disponibles.

Seleccionados=S=Las cajas de bombones que ya hemos seleccionado.

Función solución=F=Será solución cuando el dinero se agote.

Función objetivo=Minimizar numero de cajas distintas.

Función de factibilidad= Es factible si aun queda espacio y no esta en el conjunto de seleccionados (S).

Función de selección=Caja que no hayamos seleccionado.

Dinero= Será el dinero disponible.

```
ForrestGump(C,Dinero,S){
        S=0;
        caja=primeraCaja(C);
        while(C!=0 && Dinero>=caja.precio){
                if(Factible(caja,C)){
                        S=caja;
                        Dinero=Dinero-caja.precio;
                }
                caja=SeleccionarCaja(C);
        }
        return S;
}
```

98- Un repartidor de pizzas tiene que distribuir *N* encargos, que le han asignado previamente, y tratará de hacerlo en el menor tiempo posible, porque desea "pegarle a la hebra" con la vecina del quinto. Para ello, se ha provisto de un mapa de la ciudad, en el que ha señalado las casas a las que tiene que llevar encargos (un encargo por casa), y en el que ha anotado la distancia, en tiempo, entre casas a las que tiene que llevar encargos, y también la distancia (en tiempo) de la central de reparto hasta cada casa. Supuesto que en cada viaje puede llevar como mucho *M* pizzas, diseñe un algoritmo que resuelva el problema utilizando una estrategia basada en *Programación Dinámica* o *Backtracking*.

El espacio de búsqueda lo vamos a representar con un árbol combinatorio. En el que cada nivel representará la decisión de a que casa ir o si volver a la central. La solución se representa con un vector s, donde s[i] es la decisión tomada para ese momento. En el vector habrá valores desde 0 a N, siendo 0 la decisión de volver a central.

```
vector<int> pizzero(const vector<Encargo> & encargos, int capacidadMoto){
        vector<int> s(objs.size()-1), sOptimo;
        int nivel=0;
        int m=capacidadMoto;
        while(nivel>=0){
                s[nivel]++;
                if(s[nivel]==encargos.size()){
                        nivel--;
                        s[nivel]==-1;
                }
                else{
                        if(factible(s,encargos)){
                                if(nivel==objs.size()){
                                        if(beneficio(s)>beneficio(sOptimo)) sOptimo=s;
                                }
                                else nivel++;
                        }
                }
        }
        return s;
}
```

1.− Aplique el algoritmo dado para el problema de la mochila en los siguientes casos:

a) M = 15, V = [10 5 15 7 6 18 3]

P = [2 3 5 7 1 4 1].

b) Seleccione un lote de 100 libros, lo más barato posible, de entre los siguientes lotes

Lote de 25 volúmenes 65.000 Ptas./Lote

Lote de 10 volúmenes 30.000 Ptas./Lote

Lote de 100 volúmenes 270.000 Ptas./Lote

Lote de 80 volúmenes 160.000 Ptas./Lote

a) Vamos eligiendo en cada caso el objeto con mayor beneficio precio/peso de entre los objetos que queden.

Beneficios=(5, 1'6, 3, 1, 6, 4'5, 3)

Ordenados todos los vectores por el vector Beneficios:

Beneficios=(6, 5, 4'5, 3, 3, 1'6, 1)

V= (6, 10, 18, 3, 15, 5, 7)

P= (1, 2, 4, 1, 5, 3, 7)

Ahora construimos el vector solución=

(Siendo 1 haber metido el objeto completo, y otro número el porcentaje del objeto metido)

Solución=(1, 1, 1, 1, 0'6, 0, 0)

El beneficio total es 27'5

b)En esta ocasión ordenaremos los vectores por el precio mas bajo puesto ya que queremos minimizar el precio y para ello el beneficio es precio/nº_libros.

V=(80, 25, 100, 10)

P=(160, 65, 270, 30)

B=(2, 2'6, 2'7, 3)

Ahora construimos el vector solución=

Solución=(1, 0'8, 0, 0)

Precio total=160+52=212

2.− Construya procedimientos que resuelvan el problema de la mochila:
a) Ordenando primero los materiales (por precios).

```
void ordenarElementos(vector<int> & b, vector<int> & p){
	for(int i=0; i<b.size(); i++){
		for(int j=b.size()-1;j>=0;j++){
			if(b[j-1]>b[j]){
				swap(b[j-1], b[j]);
				swap(p[j-1], p[j]);
			}
		}
	}
}

void Mochila(int m, vector<int> & b, vector<int> & p, vector<double> & x){
	for(int i=0;i<b.size();i++){
		x[i]=0;
	}
	int pesoUsado=0,  i=0;
	ordenarElementos(b,p);
	while(pesoUsado<m){
		if( (pesoUsado+p[i])<=m){
			x[i]=1;
			pesoUsado+=peso[i];
		}
		else{
			x[i]=(m+pesoUsado)/p[i];
			pesoUsado=m;
		}
	}
}
```