UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA	**Programación III** Códigos de asignatura: Sistemas: **402048**; Gestión: **41204-** **Prueba Presencial**	**Primera Semana** Febrero 2005 Duración: **2 horas** Material permitido: **NINGUNO**

Apellidos: _____ **DNI:** _____

Nombre: _____ **Centro Asociado en el que entregó la práctica:** _____

Cuestión 1 (2 puntos).

Se dispone de un conjunto de ficheros $f_1, f_2, ..., f_n$ con tamaños $l_1, l_2, ..., l_n$ y de un disquete con una capacidad total de almacenamiento de $d<l_1+l_2+...+l_n$. a) Suponiendo que se desea maximizar el número de ficheros almacenados y que se hace uso de un planteamiento voraz basado en la selección de ficheros de menor a mayor tamaño, ¿el algoritmo propuesto siempre obtendría la solución óptima?. En caso afirmativo demostrar la optimalidad y en caso negativo ponga un contraejemplo; b) En caso de que quisiéramos ocupar la mayor cantidad de espacio en el disquete independientemente del número de ficheros almacenados, ¿una estrategia voraz basada en la selección de ficheros de mayor a menor tamaño obtendría en todos los casos la solución óptima?. En caso afirmativo demuestre la optimalidad, en caso negativo ponga un contraejemplo.

Solución:

Apartado A

El algoritmo voraz obtendría la solución óptima, es decir, un disquete con el mayor número posible de ficheros.

Demostración: Suponiendo que los programas se encuentran inicialmente ordenados de menor a mayor tamaño, vamos a demostrar la optimalidad de la solución comparando una solución óptima con una solución obtenida por el algoritmo voraz. Si ambas soluciones no fueran iguales, iremos transformando la solución óptima de partida, de forma que continúe siendo óptima, pero asemejándola cada vez más con la obtenida por el algoritmo voraz. Si consiguiéramos igualar ambas soluciones en un número finito de pasos, entonces podremos afirmar que la solución obtenida por el algoritmo es óptima.

Notación: Una solución cualquiera viene representada por $Z=(z_1, z_2, ..., z_n)$ donde $z_i=0$ implica que el fichero f_i no ha sido seleccionado como parte de la solución. De este modo, $\sum_{i=1}^{n} z_i$ indicará el número de ficheros seleccionados como solución al problema.

Siendo X la solución devuelta por la estrategia voraz e Y la solución óptima al problema. Supongamos que la estrategia voraz propuesta selecciona los k primeros ficheros (con $1 \le k \le n$), recordamos que los ficheros se encuentran inicialmente ordenados de menor a mayor tamaño. El fichero k+1 es rechazado puesto que ya no es posible incluir un solo fichero más. De este modo, la solución X será $(x_1, x_2, ..., x_k,..., x_n)$, donde $\forall i \in \{1..k\}.x_i = 1$ y $\forall i \in \{k+1..n\}.x_i = 0$.

Comenzando a comparar X con Y de izquierda a derecha, supongamos que $j \ge 1$ sea la primera posición donde $x_j \neq y_j$. En este caso, obligatoriamente $j \le k$ ya que en caso contrario la solución óptima incluiría todos los ficheros escogidos por la estrategia voraz y alguno más, lo que se contradice con el hecho de que los ficheros del k+1 al n se rechazan por la estrategia voraz porque no caben.

Este modo, $x_j \neq y_j$ implica que $y_i=0$, por lo que $\sum_{i=1}^{j} y_i = j-1$, que es menor que el número de ficheros seleccionados por nuestra estrategia voraz como solución al problema $\sum_{i=1}^{j} x_i = j$. Como suponemos que Y es una solución óptima,

$\sum_{i=1}^{n} y_i \geq \sum_{i=1}^{n} x_i = k$, esto significa que existe un l > k ≥ j tal que y_l = 1, es decir, existe un fichero posterior para compensar el que no se ha cogido antes. Por la ordenación impuesta a los ficheros, sabemos que $l_j \leq l_l$, es decir, que si f_l cabe en el disco, podemos poner en su lugar f_j sin sobrepasar la capacidad total. Realizando este cambio en la solución óptima Y, obtenemos otra solución Y' en la cual y'_j= 1 = x_j, y'_l=0 y para el resto y'_i = y_i. Esta nueva solución es más parecida a X, y tiene el mismo número de ficheros que Y', por lo que sigue siendo óptima. Repitiendo este proceso, podemos ir igualando los ficheros en la solución óptima a los de la solución voraz X, hasta alcanzar la posición k.

Apartado B

El algoritmo voraz no obtendría la solución óptima en todos los casos.

Contraejemplo: Supongamos la siguiente lista de ficheros con tamaños asociados:

Fichero	F1	F2	F3
Tamaño	40	30	15

Supongamos que la capacidad máxima del disquete es 45.

Aplicando la estrategia voraz propuesta, únicamente podríamos almacenar el fichero F1, ocupando 40 de los 45 de capacidad que tiene el disquete. Sin embargo, si hubiéramos seleccionado los ficheros F2 y F3 hubiera sido posible maximizar el espacio ocupado en el disco.

Cuestión 2 (2 punto).

Exponga y explique el algoritmo más eficiente que conozca para realizar una planificación de tareas con plazo fijo maximizando el beneficio.
Dada la tabla adjunta de tareas con sus beneficios (gi) y caducidades (di) asociados. Aplique paso a paso el algoritmo propuesto, suponiendo que se desea realizar una planificación en un tiempo t=5.

i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
gi	30	10	2	11	10	9	2	56	33
di	5	3	2	2	1	2	7	5	4

Solución:

Apartado A

El algoritmo más apropiado es el algoritmo **secuencia2** explicado y desarrollado en pg 240-241 del texto base y que ha sido utilizado en la práctica (Bloque 1). El alumno debe haberlo expuesto y explicado.

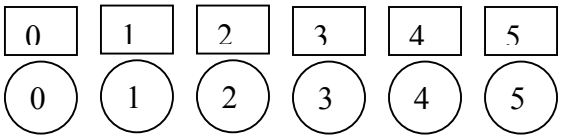
Apartado B

1. Inicialmente ordenamos la tabla propuesta por orden decreciente de beneficios.
2. Creamos el número apropiado de estructuras de partición. p=min(9,max(di))=min(9,7)=7. Como en este caso concreto queremos únicamente planificar las 5 primeras tareas p puede reducirse a 5.
3. Vamos extrayendo cada una de las tareas por orden decreciente de beneficios e incluyéndolas en su correspondiente estructura de partición. Esta operación implica fusionar la estructura en la cual se ha incluido la tarea con la estructura de partición inmediatamente anterior.
4. El algoritmo termina cuando ya no queda ninguna estructura de partición libre para asignar tarea.

i	**8**	**9**	**1**	**4**	**5**	**2**	**6**	**3**	**7**
gi	56	33	30	11	10	10	9	2	2
di	5	4	5	2	1	3	2	2	7

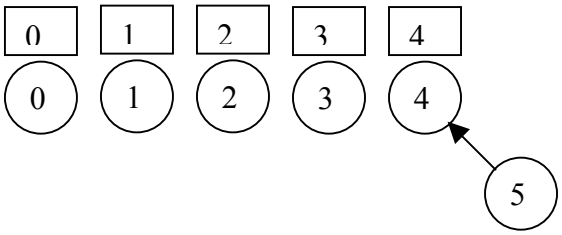
Tabla de costes y caducidades ordenada

El proceso puede esquematizarse del siguiente modo:



Res[]

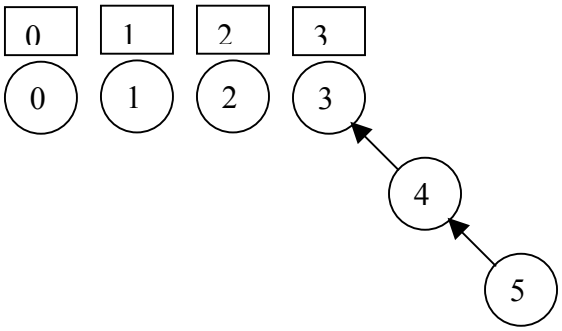
0	1	2	3	4



Selecciono tarea 8

Res[]

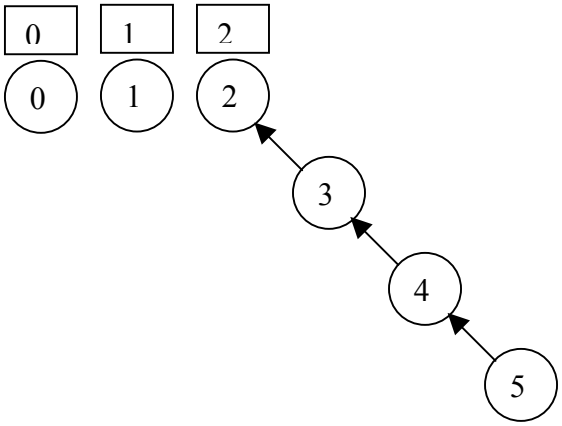
0	1	2	3	4
				8



Selecciono tarea 9

Res[]

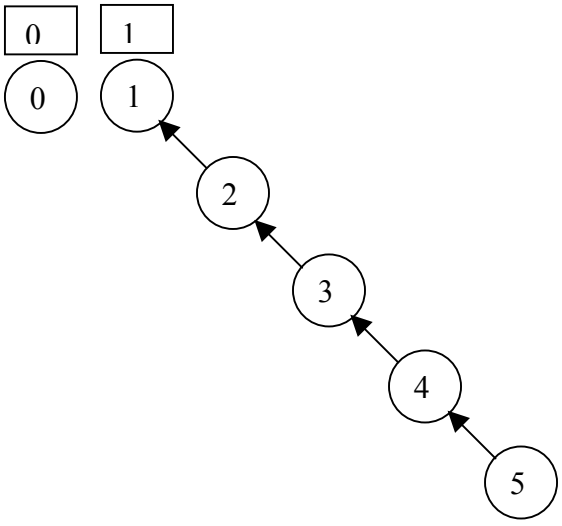
0	1	2	3	4
			9	8



Selecciono tarea 1

Res[]

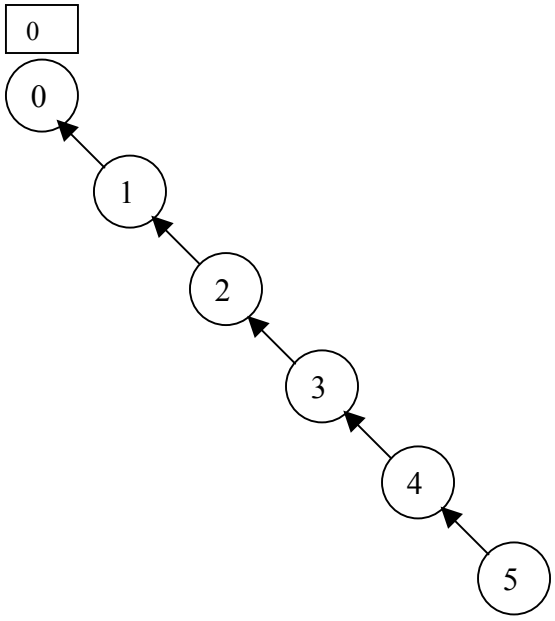
0	1	2	3	4
		1	9	8



Selecciono tarea 4

Res[]

0	1	2	3	4
	4	1	9	8



Selecciono tarea 5

	0	1	2	3	4
Res[]	5	4	1	9	8

Cuestión 3 (1 punto).
Explique las diferencias entre un recorrido en anchura y un recorrido en profundidad. Exponga un algoritmo iterativo para cada uno de los recorridos explicando las estructuras de datos asociadas, así como su coste.

Solución:

En las páginas 338 y 339 del texto base puede encontrarse una explicación tanto de las diferencias entre ambos tipos de recorrido, como un algoritmo iterativo para cada uno de ellos basado en el uso de PILAS y COLAS como estructuras de datos asociadas.

Problema (5 puntos).

Partiendo de un conjunto N={n_1, n_2, ..., n_m} compuesto por m número positivos y de un conjunto O={+,-,*,/} con las operaciones aritméticas básicas, se pide obtener una secuencia de operaciones factible para conseguir un número objetivo P. Como restricciones al problema, debe tenerse en cuenta que: a) los números del conjunto N pueden utilizarse en la secuencia de operaciones 0 o 1 vez, b) los resultados parciales de las operaciones pueden utilizarse como candidatos en operaciones siguientes, c) las operaciones que den como resultado valores negativos o números no enteros NO deberán tenerse en cuenta como secuencia válida para obtener una solución. Diseñe un algoritmo que obtenga una solución al problema propuesto, mostrando la secuencia de operaciones para obtener el número objetivo P. En caso de no existir solución alguna, el algoritmo deberá mostrar la secuencia de operaciones que dé como resultado el valor más próximo, por debajo, del número objetivo P. Por ejemplo, siendo P=960 y N={1,2,3,4,5,6}, la secuencia de operaciones que obtiene la solución exacta es: ((((6*5)*4)*2)*(3+1))=960. Si P=970, el algoritmo no encontraría la solución exacta con el conjunto de números inicial y la secuencia más próxima por debajo de P sería ((((6*5)*4)*2)*(3+1))=960.

Solución:

Elección Razonada del Esquema Algorítmico.

- Obviamente no se trata de un problema que pueda ser resuelto por divide y vencerás, puesto que no es posible descomponer el problema en subproblemas iguales, pero de menor tamaño, que con algún tipo de combinación nos permita encontrar la solución del problema.
- Tampoco se trata de un algoritmo voraz, puesto que no existe ninguna manera de atacar el problema de manera directa que nos lleve a la solución sin necesidad de, en algún momento, deshacer alguna de las decisiones tomadas.
- Por tanto, se trata de un problema de exploración de grafos donde deberemos construir el grafo implícito al conjunto de operaciones posibles con el fin de encontrar una solución al problema. Descartamos una exploración ciega en profundidad o en anchura puesto que, como en la mayor parte de los casos va a existir por lo menos una solución y además el enunciado del problema solicita una solución al problema y no todas, es deseable que la exploración se detenga en el momento en el que encuentre alguna de ellas. Sólo en el caso de que no exista solución al problema, a partir del conjunto N inicial, el recorrido deberá ser completo. De acuerdo a este razonamiento, la estrategia más apropiada parece la de aplicar un esquema del tipo backtracking o vuelta atrás. Como el alumno ya conoce, este tipo de algoritmos se basan en un recorrido en profundidad o en anchura que no construye el grafo implícito de manera exhaustiva, puesto que dispone de condiciones de corte que lo detienen en cuanto se encuentra una solución. En nuestro caso, además, basaremos el algoritmo en un recorrido en profundidad y no en anchura puesto que en la mayor parte de las ocasiones es necesario combinar prácticamente la totalidad de los elementos de N para obtener la solución. La alternativa de aplicar un algoritmo de ramificación y poda no es válida en este caso pues este tipo de algoritmos se caracteriza por obtener la solución óptima a un problema concreto. En este caso no es necesario optimizar absolutamente nada, pues la solución es el número objetivo que nos solicitan y no van a existir, por tanto, soluciones mejores o peores. Sólo en el caso de que no exista solución, el problema nos pide la más aproximada por debajo, lo cual implica una exploración exhaustiva del grafo implícito y, por tanto, el conocimiento por parte del algoritmo de cual ha sido la operación más aproximada realizada hasta el momento.

Descripción del Esquema Algorítmico de Vuelta Atrás.

```
fun vuelta-atrás(e: ensayo)
        si valido(e) entonces
                dev e
        sino
                listaensayos ← complecciones(e)

                mientras ¬ vacia(listaensayos) ∧ ¬resultado hacer
                        hijo ← primero(listaensayos)
                        listaensayos ← resto(listaensayos)

                        si condicionesdepoda(hijo) entonces
                                resultado ← vuelta-atrás(hijo)
                        fsi
                fmientras
                dev resultado
        fsi
ffun
```

Estructuras de Datos Necesarias

La estructura de datos principal para llevar a cabo nuestro algoritmo va a ser aquella encargada de almacenar la información relativa a cada uno de los ensayos generados:

```
Tipo Tensayo=
        candidatos: vector de int
        operaciones: vector de TpilaOperaciones
        solucion: boolean
        vacio: boolean

Operaciones Asociadas
---------------------

getCandidatos():vector
        Devuelve el vector de candidatos para operar con él.
getOperaciones():vector
        Devuelve el vector de operaciones(pilas) para operar con él.
getCandidato(indexCand int):int
        Devuelve el candidato que se encuentra en la posición indexCand.
removeCandidato(indexCand int):void
        Elimina el candidato que se encuentra en la posición indexCand.
setCandidato(candidato int,indexCand int):void
        Inserta el candidato candidato en la posición indexCand del vector de candidatos.
getOperacion(indexOp int):TPilaOperaciones
        Devuelve la operación(pila) que se encuentra en la posición indexOp.
removeOperacion(indexOp int):void
        Elimina la pila de operaciones que se encuentra en indexOp.
setOperacion(operacion TpilaOperaciones, indexOp int):void
        Inserta la pila de operaciones operación en la posición indexOp del vector de operaciones.
setSolucion(solucion boolean):void
        Marca el ensayo como solución válida.
isSolucion():boolean
        Devuelve un boolean que indica si el ensayo es o no solución.
isVacio():bolean
        Devuelve un bolean que indica si el ensayo es o no vacio.
setVacio(vacio boolean)void
        Marca el ensayo como vacio.
```

```
Tipo TpilaOperaciones=
        pila: pila de String

Operaciones Asociadas
---------------------

pushNumber(value int):void
        Añade un número a la pila. La lógica de la operación transforma el entero de entrada en una
        cadena de caracteres para poder insertarlo en la pila.
pushOperator(oper char):void
        Añade un operador a la pila. La lógica de la operación transforma el entero de entrada en una
        cadena de caracteres para poder insertarlo en la pila.
```

El principal problema del ejercicio se encuentra en determinar cómo vamos a ir construyendo el grafo implícito y cómo vamos a representar las operaciones que ya han sido llevadas a cabo. Para ello vamos a tener en cuenta lo siguiente:

1. Nuestro algoritmo siempre va a trabajar sobre un conjunto de candidatos que recogerá inicialmente los valores asociados y, posteriormente, los valores obtenidos al ir realizando cada una de las operaciones. Esta es la función del elemento candidatos de Tensayo.
2. Para poder recordar qué operaciones se han llevado a cabo y mostrárselas al usuario al fin del algoritmo, es necesario desplegar un almacén de información paralelo a candidatos que, para cada uno de los candidatos recoja las operaciones que éste ha soportado hasta el momento. Con este fin se crea el elemento operaciones, que no es más que un vector donde cada elemento representa una pila de operaciones y operandos que, en notación postfija, representan el historial de operaciones de dicho elemento.
3. Finalmente, el elemento solución marca el ensayo como solución o no, dependiendo si alberga la solución al problema.

Veamos con un ejemplo el funcionamiento de esta estructura de datos. Supongamos que nuestro conjunto inicial es N={1, 2, 3, 4}. El ensayo correspondiente a esta situación inicial vendría descrito por:

```
Tipo Tensayo=
        candidatos: vector de int
        operaciones: vector de TpilaOperaciones
        solucion: bolean
```

```
ENSAYO
------
candidatos:<1,2,3,4>

operaciones  <[1] [2] [3] [4] >

solucion: false
```

Supongamos ahora que operamos el candidato 2 y el candidato 4 con el operador ‘+‘. El nuevo formato del ensayo sería el siguiente:

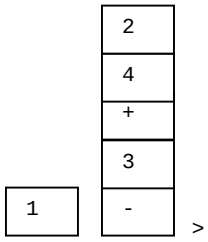
```
ENSAYO
------
candidatos:<1,6,3>
                      [2]
                      [4]
operaciones  <[1]     [+]   [3] >

solucion: false
vacio: false
```

Nótese que:

1. Desaparece el elemento que ocupa la posición 3 del vector y su pila asociada.
2. El vector de candidatos ahora almacena el valor 6 allí donde se ha realizado la operación.
3. El vector de operaciones ha actualizado la pila de operaciones, reflejando la nueva situación en notación postfija (2,4,+)

Supongamos que ahora operamos el candidato 6 con el candidato 3 utilizando el operador ‘-‘. El nuevo ensayo quedaría:

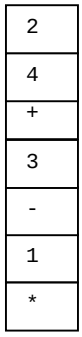
```
ENSAYO
------
candidatos:<1,3>
```




```
solucion: false
vacio: false
```

Finalmente si operamos el candidato 1 con el candidato 3 utilizando el operador ‘*’. El ensayo obtenido sería:

```
ENSAYO
------
candidatos:<3>
```




```
operaciones   <  >

solucion: false
vacio: false
```

Como podemos observar, el valor final obtenido combinando todos los valores iniciales, y de acuerdo a las operaciones descritas, sería 3 y el historial completo de todas las operaciones realizadas podría mostrarse deshaciendo la pila en formato postfijo generada (2,4,+,3,-,1,*).

Algoritmo Completo.

```
fun vuelta-atrás(e: Tensayo):Tensayo
	si valido(e) entonces
		solucion ← e
		solucion.setSolucion(true);
		dev solucion
	sino
		listaensayos ← complecciones(e)

		mientras ¬ vacia(listaensayos) ∧ ¬solucion.isSolucion() hacer
			hijo ← primero(listaensayos)
			listaensayos ← resto(listaensayos)

			si ¬podar(hijo) entonces
				solucion ← vuelta-atrás(hijo)
			sino
				solucion ← mejor(e)
			fsi
		fmientras
		dev solucion
	fsi
ffun
```

Nótese que **solución** es un ensayo, que inicialmente es vacío, y que contendrá la solución en caso de existir o la serie de operaciones que más se aproximen en caso de que esto no ocurra. Solución se define externamente a la función vuelta-atrás y, por eso, puede manipularse desde cualquiera de las funciones sin ser necesario enviarla a éstas como parámetro.

Igualmente, la el valor objetivo P, también es utilizado globalmente por la función vuelta atrás.

Las funciones asociadas al algoritmo son las siguientes:

valido(e Tensayo):boolean
Función que devuelve true si el ensayo que recibe como parámetro es solución al problema, es decir, si contiene algún candidato cuyo valor sea P. Devuelve false en caso contrario.

complecciones(e Tensayo):lista
Función que devuelve la lista de hijos correspondientes a un ensayo concreto. La política de generación de hijos que seguiremos será la siguiente: Para cada candidato, complecciones genera todas las combinaciones posibles de éste con cada uno de los demás, haciendo uso del conjunto de operaciones posibles.

podar(e Tensayo):bolean
Función que devuelve un boolean dependiendo si es posible continuar explorando por el ensayo **e** que recibe como parámetro o no. La única condición de poda que impondremos será que alguno de los candidatos calculados hasta el momento sobrepase el valor de P.

mejor(e Tensayo):Tensayo
Función que compara el ensayo **e**, que recibe como parámetro, con la solución calculada hasta el momento. Devuelve a la salida aquel ensayo que contenga el candidato más próximo a la solución solicitada.

```
fun valido(e: Tensayo):boolean
	para c desde 0 hasta numCandidatos hacer
		candidato ← e.getCandidato(c)
		si candidato = P entonces
			dev true
		fsi
	fpara
	dev false
ffun

fun podar(e: Tensayo):boolean
	para c desde 0 hasta numCandidatos hacer
		candidato ← e.getCandidato(c)
		si candidato > P entonces
			dev true
		fsi
	fpara
	dev false
ffun
```

```
fun mejor(e: Tensayo):Tensayo
	v1 ← valorMax(e)
	v2 ← valorMax(solucion)

	si v1<v2 entonces
		dev solucion
	sino
		dev e
	fsi
ffun

fun valorMax(e: Tensayo):int
	value ← e.getCandidato(0)

	para cada c desde 1 hasta numCandidatos hacer
		valAux ← e.getCandidato(c)

		si (valAux>value) ∧ (valAux<=P) entonces
			value=valAux
		fsi

	fpara

	dev value
ffun


fun complecciones(e: Tensayo):vector

	para c1 desde 0 hasta numCandidatos hacer
		para c2 desde c1+1 hasta numCandidatos hacer
			hijo=obtieneHijo(e,'+',c1,c2)
			si (¬hijo.isVacio()) entonces
				vHijos.addElement(hijo)
			fsi

			hijo=obtieneHijo(e,'-',c1,c2)
			si (¬hijo.isVacio()) entonces
				vHijos.addElement(hijo)
			fsi

			hijo=obtieneHijo(e,'*',c1,c2)
			si (¬hijo.isVacio()) entonces
				vHijos.addElement(hijo)
			fsi

			hijo=obtieneHijo(e,'/',c1,c2)
			si (¬hijo.isVacio()) entonces
				vHijos.addElement(hijo)
			fsi

		fpara
	fpara

	dev vHijos
ffun
```

```
fun obtieneHijo(e: Tensayo, char operator, int c1Index, int c2Index):Tensayo

        c1 ← e.getCandidato(c1Index)
        c2 ← e.getCandidato(c2Index)

        nuevoEnsayo ← e

        si (operator='+') entonces
                res ← c1+c2
        sino
                si (operator='-') entonces
                        res ← c1-c2
                sino
                        si (operator='*') entonces
                                res ← c1*c2
                        sino
                                si (operator='/') entonces
                                        si (c2!=0) ∧ (c1%c2=0) entonces
                                                res ← c1/c2
                                        sino
                                                res ← -1
                                        fsi
                                fsi
                        fsi
                fsi
        fsi

        si (res >=0) entonces

                nuevoEnsayo ← e

                pila1=e.getOperacion(c1)
                pila2=e.getOperacion(c2)

                pila=generaNuevaPila(pila1,pila2,operator)

                nuevoEnsayo.removeCandidato(c2)
                nuevoEnsayo.setCandidato(res,c1)
                nuevoEnsayo.removeOperacion(c2)
                nuevoEnsayo.setOperacion(pila,c1)

                dev nuevoEnsayo
        sino
                dev ensayoVacio
        fsi

ffun
```

Nótese como la funcion generaNuevaPila recibe como parámetros las dos pilas ya existentes (pertenecientes a cada uno de los candidatos), así como el operador que va a ser utilizado para combinar ambos candidatos y genera como resultado la nueva pila correspondiente al candidato generado.