Práctica 5:
Algoritmos de Programación Dinámica

ABEL RÍOS GONZÁLEZ 20%
SALVADOR ROMERO CORTÉS 20%
RAÚL DURÁN RACERO 20%
MANUEL CONTRERAS ORGE 20%
ALBERTO PALOMO CAMPOS 20%

ÍNDICE

Asignación de recursos

Se tienen r unidades de un recurso (monetarias, personal, ...) que deben asignarse a n proyectos. Si se asignan j, 0 <= j <= r, unidades al proyecto i, se obtiene un beneficio N(i, j) >= 0.
Diseñad e implementad un algoritmo de Programación Dinámica que asigne recursos a los n proyectos maximizando el beneficio total obtenido.
Aplicadlo para un problema con n = 4 proyectos y r = 6 recursos y una matriz de beneficios N(i, j).

i/j	0	1	2	3	4	5	6
1	0	1	1	2	3	4	5
2	0	2	3	3	3	4	6
3	5	0	1	1	2	2	2
4	0	2	3	4	5	6	7

Planteamiento de la solución como una sucesión de decisiones

Podemos plantear la solución como una sucesión de decisiones x1, x2, ..., xn donde en cada paso se decide cuántas unidades de un recurso se asignan al proyecto 1, cuántas al proyecto 2, y así sucesivamente hasta asignar todos los proyectos…

Verificación del principio de optimalidad

Partimos de una secuencia de $x_1, x_2, x_3 ... x_n$ que representa las decisiones de asignar los recursos de manera que x_i muestra el número de recursos que se le asignan al proyecto i . Definimos esta secuencia como la solución óptima con n proyectos y r recursos. De esta manera, la secuencia cumple:

- $\sum_{1}^{n} x_i \leq r$
- $\sum_{1}^{n} N(i, x_i)$ es máximo. Siendo $N(i, x_i)$ el beneficio obtenido con la decisión x_i ; esto es asignar los x_i recursos de esa decisión al proyecto i.

Vamos a demostrar por contradicción que la subsecuencia $x_1, x_2, x_3, ...x_{n-1}$ es óptima para un subproblema con $n-1$ proyectos y $r-k$ recursos.

Si la subsecuencia x no fuera óptima existiría una subsecuencia y que sí sería la óptima para $n-1$ proyectos. De esta manera:

- $\sum_{1}^{n-1} y_i \leq r-k$
- $\sum_{1}^{n-1} N(i, y_i) > \sum_{1}^{n-1} N(i, x_i)$

Pero entonces, haciendo $y_n = x_n$ tenemos que:

- $\sum_{1}^{n-1} y_i + k \leq r$
- $(\sum_{1}^{n-1} N(i, y_i)) + N(n, y_n) > (\sum_{1}^{n-1} N(i, x_i)) + N(n, x_n)$

Con esto último llegamos a una contradicción con la hipótesis planteada, puesto que habríamos encontrado una sucesión $y_1, y_2, y_3, ...y_n$ que sería mejor que la que hemos definido como óptima. Esta contradicción significa que la hipótesis planteada de que la subsecuencia de los n-1 elementos no es óptima para el subproblema de n-1 proyectos es errónea, y que por lo tanto la subsecuencia sí sería óptima para el subproblema.

Definición recursiva de la solución óptima

```
n = número de proyectos
r = número de recursos

si n = 0 entonces:
      B[n,r] = 0
para k entre 0..r:
      B[n,r] = max{ B[n-1,r-k] + N[n,k] } // máximo entre todas las
iteraciones
```

Siendo B[n,r] el beneficio obtenido con n proyectos y r recursos, y N(n,k) el beneficio de asignar k recursos al proyecto número n.

Cálculo del valor de la solución óptima usando un enfoque ascendente

```
n = número de proyectos
r = número de recursos
M = matriz que almacena los cálculos (memoización)
Ar = matriz que almacena los valores de recursos usados para el max
beneficio

si n = 0 entonces:
    return 0
si B[n,r] está en la matriz M
    return M[n,r]
en otro caso:
    M[n,r] = -infinito
    para k entre 0..r:
          anterior = M[n,r]
          M[n,r] = max{B[n-1,r-k] + N[n,k] , M[n,r]}
          si anterior != M[n,r] entonces:
              Ar[n,r] = k

    return M[n,r]
```

Para ahorrar cálculos usamos una matriz M que almacena en cada posición [i,j] el beneficio máximo obtenido para esa combinación de i proyectos con j recursos. En cada iteración comprobamos si ya hemos calculado ese dato o no. En caso de que sea la primera vez que accedemos a esta celda se le calcula el valor.
Además usamos una matriz auxiliar Ar que almacena los datos del número de recursos usado. Esta matriz nos servirá más tarde para determinar cuál es la asignación óptima.

Determinación de la solución óptima empleando la información almacenada en la tabla

Una vez ya hemos determinado cuál es el beneficio máximo y hemos conseguido rellenar la tabla Ar, mencionada anteriormente, obtener la asignación óptima resulta trivial, puesto que en esta tabla se almacena en la posición [i,j] el número de recursos que se le debe asignar a ese proyecto i con los j recursos que tenga disponible. De manera que comenzamos en la esquina inferior derecha, en la que tenemos el número de de recursos que le asignamos al proyecto 4. A continuación, subimos una fila y nos desplazamos a la izquierda tantas columnas como recursos utilizó la fila anterior. Esto lo vamos repitiendo hasta agotar el número de filas. El algoritmo para determinar esta solución sería así:

```
vector<int> asignacion //vector que almacena la solución
desplazamiento = recursos
para i entre proyectos..0:
    asignacion[i] = Ar[i,desplazamiento]
    desplazamiento = desplazamiento - Ar[i,desplazamiento]
return asignacion
```

La deducción de este algoritmo se basa en hacer el proceso inverso de la recurrencia del beneficio.

Algoritmo

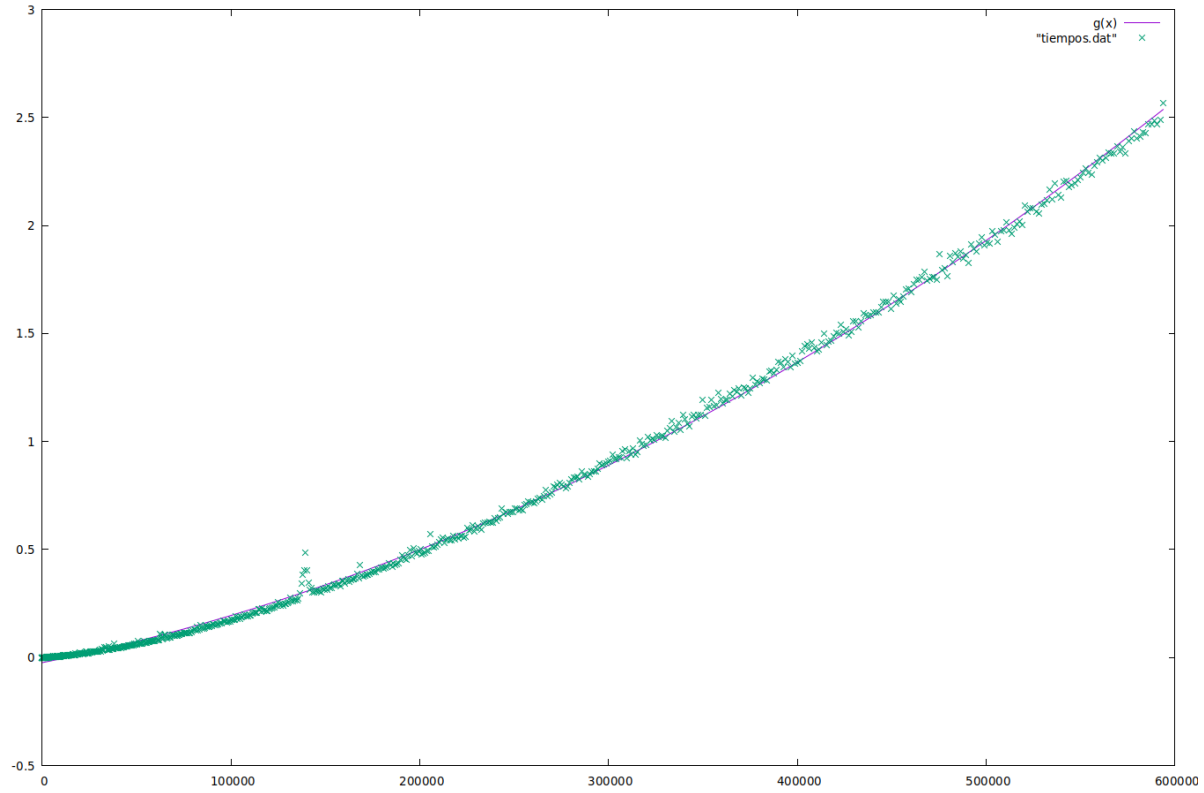
Tras todos los pasos anteriores el algoritmo final será:

```
n = número de proyectos
r = número de recursos
M = matriz que almacena los cálculos (memoización)
Ar = matriz que almacena los valores de recursos usados para el max
beneficio

funcion B[n,r]:
    si n = 0 entonces:
        return 0
    si B[n,r] está en la matriz M
        return M[n,r]
    en otro caso:
        M[n,r] = -infinito
        para k entre 0..r:
              anterior = M[n,r]
              M[n,r] = max{B[n-1,r-k] + N[n,k] , M[n,r]}
              si anterior != M[n,r] entonces:
                  Ar[n,r] = k
        return M[n,r]
//cálculo de la asignación óptima
función calcularAsignacion:
    vector<int> asignacion //vector que almacena la solución
    desplazamiento = recursos
    para i entre proyectos..0:
        asignacion[i] = Ar[i,desplazamiento]
        desplazamiento = desplazamiento - Ar[i,desplazamiento]
    return asignacion
```

Eficiencia del algoritmo:
No sabemos con certeza la eficiencia teórica exacta del algoritmo, pero, al contrastar con los datos que hemos obtenido de forma empírica, todo parece apuntar a una eficiencia del orden O(nr^2).

Pruebas empíricas:



Ejemplo

En este ejemplo usaremos los siguientes datos:
Número de proyectos n = 4
Número de recursos r = 6

i/j	0	1	2	3	4	5	6
1	0	1	1	2	3	4	5
2	0	2	3	3	3	4	6
3	5	0	1	1	2	2	2
4	0	2	3	4	5	6	7

Debido a que el tamaño del ejemplo es muy grande como para hacerlo completo manualmente, realizaremos las primeras iteraciones y el cálculo final de la asignación.

Empezamos llamando a la función B(4,6). La tabla actualmente está vacía, por lo que se marca como -infinito y se llama a B(3,6). Se vuelve a marcar como -infinito. Continuamos con la llamada recursiva hasta B(0,6) que devolverá 0+N(1,6) = 5; esto es el máximo beneficio posible que tiene con los 6 recursos, que se ha obtenido recorriendo un bucle desde 0 recursos hasta 6. Por tanto, se produce la primera actualización de la tabla:

M[i,j]	0	1	2	3	4	5	6
1							5
2							
3							
4							

Ar[i,j]	0	1	2	3	4	5	6
1							6
2							
3							
4							

Así que la siguiente llamada será la de B(2,6) que llamará a B(1,5):

M[i,j]	0	1	2	3	4	5	6
1						4	5
2							6
3							
4							

Ar[i,j]	0	1	2	3	4	5	6
1						5	6
2							1
3							
4							

A continuación se calculará B(1,4).

M[i,j]	0	1	2	3	4	5	6
1					3	4	5

2							6
3							
4							

Ar[i,j]	0	1	2	3	4	5	6
1					4	5	6
2							1
3							
4							

Luego B(1,3), así sucesivamente hasta rellenar la primera fila (además de las llamadas anteriores que llamaban a B(1,x). Después de todo esto quedará así:

M[i,j]	0	1	2	3	4	5	6
1	0	1	1	2	3	4	5
2							6
3							
4							

Ar[i,j]	0	1	2	3	4	5	6
1	0	1	1	3	4	5	6
2							1
3							
4							

Luego se actualiza el valor de B(3,6):

M[i,j]	0	1	2	3	4	5	6
1	0	1	1	2	3	4	5

2							6
3							11
4							

Ar[i,j]	0	1	2	3	4	5	6
1	0	1	1	3	4	5	6
2							1
3							0
4							

Si seguimos, entonces se rellenará la fila de 2 proyectos (al cabo de unas cuantas llamadas):

M[i,j]	0	1	2	3	4	5	6
1	0	1	1	2	3	4	5
2	0	2	3	4	4	5	6
3							11
4							

Ar[i,j]	0	1	2	3	4	5	6
1	0	1	1	3	4	5	6
2	0	1	1	2	1	1	1
3							0
4							

Luego la tercera fila:

M[i,j]	0	1	2	3	4	5	6
1	0	1	1	2	3	4	5

2	0	2	3	4	4	5	6
3	5	7	8	9	9	10	11
4							13

Ar[i,j]	0	1	2	3	4	5	6
1	0	1	1	3	4	5	6
2	0	1	1	2	1	1	1
3	0	0	0	0	0	0	0
4							13

Así finaliza el algoritmo pues ya ha obtenido el valor de B(4,6) = 13.

Ahora necesitamos recuperar cuál ha sido la asignación que ha provocado este resultado: Tendremos el vector de asignaciones vacío:

v[1]	**v[2]**	**v[3]**	**v[4]**

Y comenzando desde la esquina inferior derecha vamos obteniendo los valores de las asignaciones:

v[1]	**v[2]**	**v[3]**	**v[4]**
			3

Luego, realizando la recurrencia al revés, avanzamos una fila hacia arriba y v[3] hacia la izquierda:

v[1]	**v[2]**	**v[3]**	**v[4]**
		0	3

Repetimos dos veces más:

v[1]	**v[2]**	**v[3]**	**v[4]**
	2	0	3

v[1]	v[2]	v[3]	v[4]
1	2	0	3

Para apreciar el algoritmo de recuperación de asignación de una forma más visual y esquemática podemos plantearlo de esta forma:

Ar[i,j]	0	1	2	3	4	5	6
1	0	1	1	3	4	5	6
2	0	1	1	2	1	1	1
3	0	0	0	0	0	0	0
4							3

El rojo intenso representa el valor de la asignación y el color claro indica el desplazamiento, que coincide con el valor de la asignación de la fila de abajo.

Implementación

Para la implementación, hemos usado una clase Solución que almacena todas las matrices y datos necesarios para evitar usar variables globales y tener un mejor control sobre los datos.
La entrada de la tabla de los beneficios se hace a través de un fichero que se pasa como parámetro del programa. El esquema del fichero de beneficios es el siguiente:

```
número de proyectos
número de recursos
matriz de beneficios
```

Con el ejemplo proporcionado en el guión sería:

```
4
6
0 1 1 2 3 4 5
0 2 3 3 3 4 6
5 0 1 1 2 2 2
0 2 3 4 5 6 7
```

La implementación del algoritmo para obtener el beneficio máximo sería la siguiente:

```
int Solucion::B(int n, int r) {

   if (n == 0){
       return 0;
```

```
    }
    int prev = M[n-1][r];
    if (M[n-1][r] == -1){
        M[n-1][r] = numeric_limits<int>::min(); //"-infinito"
        for (int k=0; k <= r; k++){
            prev = M[n-1][r];
            M[n-1][r] = max(M[n-1][r],B(n-1,r-k) + N[n-1][k]);
            if (prev != M[n-1][r]){
                Ar[n - 1][r] = k;
            }
        }
    }


    return M[n-1][r];
}
```

La inicialización de la matriz M se hizo con valores -1 para poder comprobar fácilmente si esa celda se ha visitado anteriormente o no.
Para el -infinito usamos el mínimo valor que pueda tener un entero. Aunque realmente serviría poner un valor como -2 ya que todos los beneficios son mayores o iguales a 0.
Para obtener la asignación hemos ido incluyendo en la matriz Ar los valores de k asociados al mejor beneficio. Eso lo hacemos con la variable prev (previous) que comprueba si se ha actualizado el valor del máximo.
Ahora una vez hayamos calculado el máximo beneficio posible y hemos rellenado la tabla Ar, podemos calcular cuál ha sido la asignación que ha producido ese resultado:

```
std::vector<int> Solucion::getAsignacion() {
    int desplazamiento = recursos-1;
    for (int i=proyectos-1; i >= 0; i--){
        asignacion[i] = Ar[i][desplazamiento];
        desplazamiento -= Ar[i][desplazamiento];
    }
    return asignacion;
}
```

Se trata de la implementación del algoritmo comentado anteriormente. Desplazarse a través de la matriz e ir asignando los valores correspondientes a cada proyecto. El vector asignación representa el número de proyectos asignados a cada proyecto en esa posición. De manera que si `asignacion[2] = 3`, querrá decir que al proyecto 3 (al vector se accede desde el 0) se le asignan 3 recursos.

Si le pasamos el archivo de ejemplo al programa este nos devuelve el siguiente resultado:

```
┌─<salva @ pop-os in ~/U/2/A/PD>
└─< (main)* >──» ./PD ejemplo.txt

BENEFICIOS:
  0   1   1   2   3   4   5
  0   2   3   3   3   4   6
  5   0   1   1   2   2   2
  0   2   3   4   5   6   7
MEMOIA:
  0   1   1   2   3   4   5
  0   2   3   4   4   5   6
  5   7   8   9   9  10  11
 -1  -1  -1  -1  -1  -1  13

Asignacion:
  0   1   1   3   4   5   6
  0   1   1   2   1   1   1
  0   0   0   0   0   0   0
 -1  -1  -1  -1  -1  -1   3

Beneficio Máximo: 13
Asignación óptima:
Al proyecto 1 se le asignan 1 recursos
Al proyecto 2 se le asignan 2 recursos
Al proyecto 3 se le asignan 0 recursos
Al proyecto 4 se le asignan 3 recursos
```

Que como vemos, es correcto y óptimo.

Para compilar basta con:

```
g++ -o PD main.cpp Solucion.cpp
```

y ejecutar con algunos de los ejemplos:

```
./PD ejemplo.txt
```