INF-155: Algoritmos y complejidad
Tarea #5
Dilema

Anghelo Carvajal
201473062-4

3 de diciembre de 2018

Carina, John y Rodrigo tienen n kilogramos de dulces ácidos que les obsequió el malvado doctor Von Brand. Ellos quieren repartirse el botín, pero como son personajes muy especiales quieren repartírselo en porciones de cierto peso que cumplan con ciertas condiciones:

- Carina solo quiere porciones que pesen a kilos, John quiere varias porciones de b kilos y Rodrigo solo aceptará porciones que tengan c kilos.
- Por supuesto, siendo una tarea de algoritmos, no quieren desperdiciar dulce alguno (El doctor siempre les dará una cantidad valida que puedan repartirse según sus condiciones) y maximizar la cantidad de porciones que cumplan con los pesos indicados.

1. Pregunta 1

Proponga y programe un algoritmo de **fuerza bruta** para resolver este problema.

1.1. Respuesta 1

Un posible algoritmo para solucionar este dilema, seria probando todos los casos posibles, de forma recursiva.

El algoritmo vería que pasaría si repartimos una porción de a kilos, otra de b kilos y otra de c kilos, quedando con 3 subproblemas (repartir $n-a$, $n-b$ y $n-c$ kilos), y de estos 3 subproblemas, nos quedamos con el que nos entregue la mayor cantidad de porciones.

Estos 3 subproblemas se plantearían como 3 llamados recursivos, de modo que llamaríamos a la función y le sumaríamos 1 al resultado, luego nos quedamos con el mayor de estos 3.

Lógicamente, no podemos repartir, por ejemplo, a kilos si tenemos $n < a$ kilos a repartir. Para este caso especial diremos que se entregaron $-\infty$ porciones, de modo que este camino se ignoraría completamente.

Todo lo explicado anteriormente se puede expresar en forma de una función matemática de la siguiente manera:

$$\phi(n,a,b,c) = \begin{cases} 0 & ; n \leq 0 \\ max \begin{cases} 1+\phi(n-a,a,b,c) & ; n-a \geq 0 \\ -\infty & ; \text{caso contrario} \\ 1+\phi(n-b,a,b,c) & ; n-b \geq 0 \\ -\infty & ; \text{caso contrario} \\ 1+\phi(n-c,a,b,c) & ; n-c \geq 0 \\ -\infty & ; \text{caso contrario} \end{cases} & ; n > 0 \end{cases}$$

La implementación en `C` del algoritmo anterior es:

Código 1: `tarea-5-1.c`

```c
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#define MAX(x, y) (((x) > (y)) ? (x) : (y))

double fuerza_bruta(long n, double a, double b, double c){
    if(0 >= n){
        return 0;
    }
    double option_a = n < a ? -INFINITY : 1 + fuerza_bruta(n-a, a, b, c);
    double option_b = n < b ? -INFINITY : 1 + fuerza_bruta(n-b, a, b, c);
    double option_c = n < c ? -INFINITY : 1 + fuerza_bruta(n-c, a, b, c);

    return  MAX(MAX(option_a, option_b), option_c);
}

int main(int argc, char **argv){
    long n;
    double a, b, c;
    int ret = scanf("%li %lf %lf %lf", &n, &a, &b, &c);
    if(ret != 4){
        exit(-1);
    }
    printf("%li\n", (long)fuerza_bruta(n, a, b, c));
    return 0;
}
```

2. Pregunta 2

Proponga y programe un algoritmo de **programación dinámica** para resolver este problema.

2.1. Respuesta 2

Es notorio que muchos de los cálculos realizados por el algoritmo anterior se repiten muchas veces, por lo cual nos aprovecharemos de eso.

Todos estos cálculos los realizaremos desde el cero hasta n (y no desde n hasta cero como en el algoritmo anterior), almacenando cada resultado. Por ende, cada vez que necesitemos realizar un calculo en base a resultados anteriores, podremos acceder a ellos de forma rápida, sin tener que calcularlos de nuevo.

Esto lo implementaríamos con un arreglo que contendría $n+1$ posiciones. Cada elemento con posición k del arreglo contendría como repartir de mejor manera k kilos de dulces, de modo que la posición n del arreglo contendría como repartir los n kilos.

El algoritmo optimizado implementado en `C` seria:

Código 2: `tarea-5-2.c`

```c
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#define MAX(x, y) (((x) > (y)) ? (x) : (y))

double programacion_dinamica(long n, double a, double b, double c){
    if(0 == n){
        return 0;
    }
    double memoization[n+1], option_a, option_b, option_c;
    memoization[0] = 0;
    for(long i = 1; i < n+1; ++i){
        option_a = i < a ? -INFINITY : 1 + memoization[(long)(i-a)];
        option_b = i < b ? -INFINITY : 1 + memoization[(long)(i-b)];
        option_c = i < c ? -INFINITY : 1 + memoization[(long)(i-c)];

        memoization[i] = MAX(MAX(option_a, option_b), option_c);
    }

    return memoization[n];
}

int main(int argc, char **argv){
    long n;
    double a, b, c;
    int ret = scanf("%li %lf %lf %lf", &n, &a, &b, &c);
    if(ret != 4){
        exit(-1);
    }
    printf("%li\n", (long)programacion_dinamica(n, a, b, c));
    return 0;
}
```

3. Pregunta 3

Haga una comparación superficial del rendimiento empírico de sus programas, muestre sus resultados y explique brevemente por qué cree que está obteniendo mejor rendimiento en uno u otro. Se evaluará lo creíble y honesta de su explicación

3.1. Respuesta 3

Para la comparación superficial, se usaron los input `50 2 3 5`, `60 2 3 5`, `65 2 3 5` y `70 2 3 5`.

Ambos programas entregaron los mismos resultados, pero tuvieron tiempos de ejecución distintos.

En la siguiente tabla se pueden ver los tiempos correspondientes:

Los tiempos fueron medidos con el programa `time` de linux.

Como se puede apreciar, el tiempo de ejecución explota con el algoritmo que usa fuerza bruta, mientras que el algoritmo que usa programación dinámica se mantenían todos pequeños y casi iguales.

input	Fuerza bruta	Programación dinámica
`50 2 3 5`	0m0.464s	0m0.006s
`60 2 3 5`	0m16.490s	0m0.007s
`65 2 3 5`	1m38.700s	0m0.007s
`70 2 3 5`	9m57.489s	0m0.010s

Cuadro 1: Comparación en los tiempos de ejecución entre algoritmos

El algoritmo de fuerza bruta toma muchísimo mas tiempo que el algoritmo de programación dinámica, debido a que para cada calculo que se quiera realizar se debe recalcular todos los otros números necesarios de forma recursiva, haciendo 3 llamados recursivos por cada cálculo, y cada uno de esos cálculos hará 3 llamados recursivos y así sucesivamente, haciendo que tome un tiempo demasiado grande.

En contraparte, el algoritmo de programación dinámica es simplemente llenar un arreglo de forma lineal, lo cual tiene baja complejidad y es muchísimo mas rápido, además de que los cálculos necesarios en cada iteración ya existen y es fácil acceder a ellos, no hay que recalcularlos cada vez.