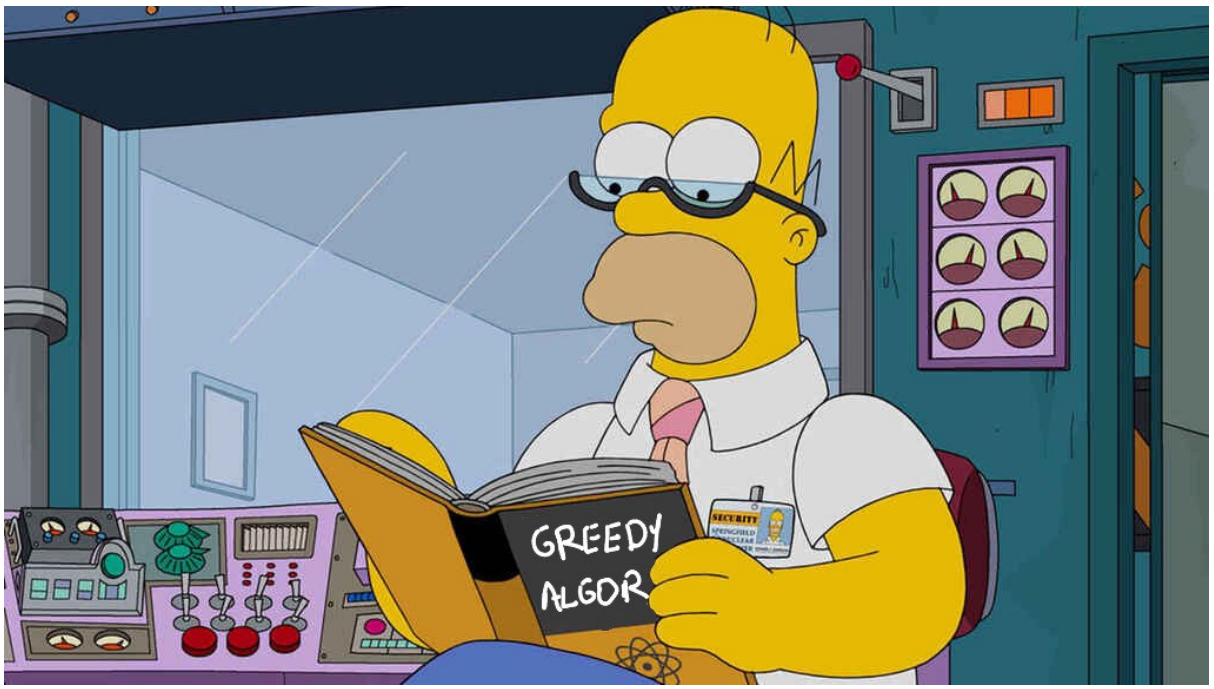
PRACTICA 2
Algoritmos voraces (Greedy)



Manuel Contreras Orge
Raúl Durán Racero
Alberto Palomo Campos
Abel Rios Gonzalez
Salvador Romero Cortés

Índice

Abastecimiento Óptimo

Una explotación ganadera necesita disponer siempre de un determinado tipo de pienso. La cantidad máxima que puede almacenar la consume en r días, y antes de que eso ocurra el ganadero necesita acudir a un almacén del pueblo para abastecerse. El problema es que dicho almacén tiene un horario de apertura muy irregular (solo abre determinados días). El ganadero conoce los días en que abre el almacén, y desea minimizar el número de desplazamientos al pueblo para abastecerse. Se pide diseñar un algoritmo greedy que determine en qué días debe acudir al pueblo a comprar pienso durante un periodo de tiempo determinado (por ejemplo durante el siguiente mes). También debemos demostrar que el algoritmo encuentra siempre la solución óptima.

Algoritmo:

```
 r = Días que tarda en consumir la capacidad máxima del almacén.

//Ordenamos los días en orden creciente:
0 = d0 < d1 < ... < dn; // Conjunto Candidatos

set<dia> S; // Conjunto de los días seleccionados

x = d0; // Posición inicial en el primer día

while(x!=dn){
      dp = día más distante siempre y cuando dp <= (x+r)
      if (dp==x) // Si el próximo día es el actual, significa que no hay
solución (el almacén del pueblo no abre antes de que se le acaben los
suministros al ganadero)
        return "No hay solución";
      else{
        x=dp; // Se selecciona el próximo día
        S.push_back(dp); // Se añade el día al conjunto de días en los
que el ganadero va a ir a comprar
      }
}

return S, S.size();
```

Demostración por **reducción al absurdo**:

Sean $0 = d0 < d1 < ... < dp = D$ los días seleccionados por el algoritmo Greedy, asumimos que D no es óptimo.

Sobre todas las soluciones óptimas $0 = h0 < h1 < ... < hq$, donde q<p, llamaremos m al máximo valor posible donde $h0 = d0$, $h1 = d1$, ... , $hm = dm$. Sea D_op una de estas soluciones, tenemos que:

1) $d(m+1) > h(m+1)$, puesto que el algoritmo greedy siempre escoge el día más lejano dentro del límite (la duración del pienso con un almacén completamente lleno), y no puede ser igual porque entonces m sería igual a m+1 respecto a la m que hemos considerado
2) Otra solución al problema sería: $0 = d0 < ... < dm < d(m+1) < h(m+2) < ... < hq$, que llamaremos X. Esto se ve claramente ya que $d(m+1)$ está más lejos de dm que $h(m+1)$, lo que provoca que d(m+1) esté más cerca de $h(m+2)$ que $h(m+1)$, por lo que forzosamente hay tambíen menos de r días entre $d(m+1)$ y $h(m+2)$, y es por lo tanto una opción válida
3) `X.size == D_op.size` -> La solución del algoritmo es óptima, ya que tiene el mismo tamaño que D_op. Esta nueva solución óptima coincide con la solución obtenida mediante el algoritmo Greedy desde el principio hasta el día m+1, lo cual es una contradicción ya que m era el número máximo de coincidencias al principio de la solución. Puesto que llegamos a una contradicción a través de un razonamiento correcto, la conclusión a la que se llega es que la premisa de que este algoritmo Greedy no encuentra la solución óptima es falsa, y entonces se puede afirmar con completa certeza que este algoritmo Greedy encuentra la solución óptima.

A modo de ejemplo, una posible implementación que hemos elaborado del algoritmo diseñado:

```
set<int>::iterator Seleccion(set<int> dias_abierto,
set<int>::iterator dia_x, int duracion_pienso, set<int>::iterator
ultimo_dia_abierto){
    set<int>::iterator dia = dia_x;

    while(*dia <= duracion_pienso + *dia_x && *dia !=
ultimo_dia_abierto){
        dia++;
    }
    if(dia != dia_x)
        dia--;

    return dia;
}
```

```
set<int> AbastecimientoOptimo(const set<int> dias_abierto, const
int duracion_pienso, int & n_dias_compra, int periodo){
    set<int> dias_compra;
    n_dias_compra = 0;
    set<int>::iterator dia_x = dias_abierto.begin();
    set<int>::iterator ultimo_dia_abierto = dias_abierto.end();
    ultimo_dia_abierto--;
    set<int>::iterator siguiente_dia;
    bool solucionable = true;

    while(dia_x != ultimo_dia_abierto && solucionable){
        siguiente_dia = Seleccion(dias_abierto, dia_x,
duracion_pienso, ultimo_dia_abierto);
        if(siguiente_dia == dia_x){
            solucionable = false;
        }
        else{
            dia_x = siguiente_dia;
            dias_compra.insert(siguiente_dia);
            n_dias_compra++;
        }
    }
    if(dia_x+10 < periodo)
        dias_compra.insert(ultimo_dia_abierto);

    return dias_compra;
}
```

Problema de los recipientes

Se pide meter n objetos de peso w_1, w_2, …, w_n, en un número ilimitado de recipientes iguales de tamaño máximo R (siendo $w_i < R$ para todo i). Los objetos se deben meter en los recipientes sin partirlos, y sin superar su capacidad máxima.

Se busca el mínimo número de recipientes necesarios para colocar todos los objetos.

Solución 1:

El primer paso de esta solución consiste en ordenar los objetos de menor a mayor (dependiendo de su peso), tarea que realizamos mediante quicksort para reducir el impacto de la ordenación en la eficiencia del algoritmo.

Una vez ordenados, introducimos los objetos en los recipientes cumpliendo las condiciones ya mencionadas. Se introducen los objetos de menor a mayor tamaño hasta que no caben más objetos en el recipiente (lo cual no quiere decir que esté lleno), y el elemento con el que se comprueba que no caben más se introduce en el siguiente recipiente. Finalmente se repite el proceso sucesivamente.

La naturaleza voraz de este algoritmo se debe a que se intenta introducir el mayor número de elementos posibles en el recipiente actual, cogiendo para ello los elementos más pequeños que asegurarán que esto se cumpla. No obstante, eso se cumplirá sobre todo al principio, ya que conforme los objetos van siendo introducidos los tamaños de los objetos serán mayores y habrá mucho espacio sobrante en los recipientes. Esto último se reflejará y tratará más adelante en la memoria.

La ventaja de usar este tipo de ordenación (menor a mayor) es que sabemos que si un objeto **i** no entra en ese recipiente, tampoco entrará ninguno que esté a su derecha ya que sabemos que será mayor o igual, por lo que debemos pasar directamente al próximo recipiente.

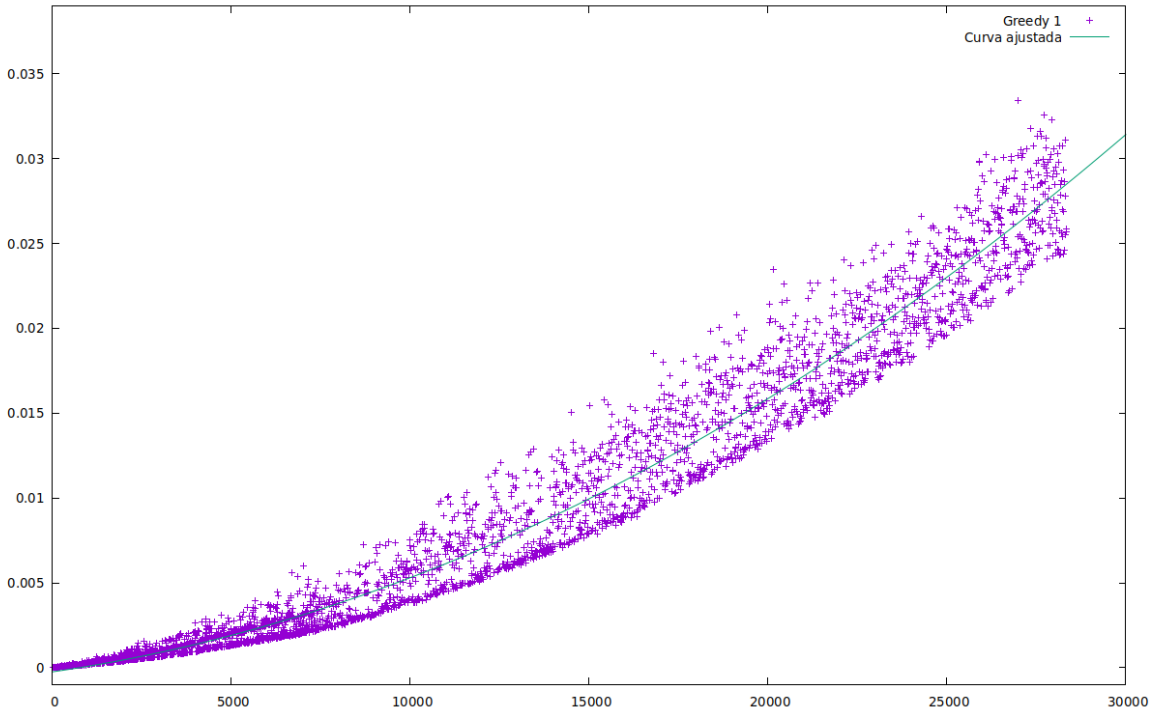
El algoritmo resultante tiene una complejidad cuadrática ($O(n^2)$). Esto proviene del bucle while que realiza n iteraciones que ejecutan funciones que son de orden O(n) en el peor caso (función erase y función push_back).

*Hacemos uso de la función **compare1** y **qsort** proporcionados por la STL para realizar la ordenación de menor a mayor*

```
int compare1 (const void * a, const void * b){
    return ( *(int*)a - *(int*)b );
}

vector<pair<float,int>> algoritmoVoraz1(){ //En nuestro caso, como los
pesos siempre son menor a R y hay infinitos recipientes, siempre hay
solución
```

```
    qsort(w, n, sizeof(float), compare1); // Ordenamos de los pesos de
los objetos el vector por quicksort

    vector<float> pesos(w,w+n);       // Vector pesos
    vector<pair<float,int>> solucion; // Vector solución con peso y
recipiente en el que se ha guardado
    float actual;                     //Peso objeto a introducir en el
recipiente
    int n_recipientes = 1;            //Numero de recipientes utilizados
    float recipiente_actual = 0.0;    //Tamaño ocupado del recipiente
actual

    while (pesos.size() != 0){
        actual = pesos.front();
        pesos.erase(pesos.begin()); // Eliminamos el objeto actual que
ya hemos introducido previamente.

        if (actual + recipiente_actual <= R){   //Si cabe en el
recipiente actual, lo insertamos
            solucion.push_back(pair<float,int>(actual,n_recipientes));
            recipiente_actual+=actual;
        }
        else {  //Si no cabe lo insertamos en el siguiente recipiente
            n_recipientes++;
            solucion.push_back(pair<float,int>(actual,n_recipientes));
            recipiente_actual = actual;
        }
    }

    return solucion;
}
```

Gráfica:



Constantes ocultas

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 2.50241e-11      +/- 4.41e-13     (1.762%)
b               = 3.04597e-07      +/- 1.228e-08    (4.033%)
c               = -0.000238952     +/- 6.556e-05    (27.44%)

correlation matrix of the fit parameters:
                a      b      c
a               1.000
b              -0.968  1.000
c               0.726 -0.843  1.000
```

Solución 2:

El primer paso de esta otra solución es ordenar los objetos de mayor a menor (dependiendo de su peso), haciendo uso nuevamente del algoritmo quicksort.

Este caso no es tan sencillo como en el anterior ya que cuando no cabe un objeto **i** en un recipiente cualquiera, no podemos afirmar que el objeto **i+1** no entrará ya que este podrá ser más pequeño y encajar en ese recipiente.

Entonces la forma de proceder es buscar sucesivamente el objeto más pesado que quepa en el recipiente o en el espacio sobrante del mismo hasta que el recipiente quede completamente lleno o no se encuentre ningún objeto que quepa.

La naturaleza voraz de este algoritmo se debe a que pretende ocupar la mayor cantidad de espacio posible del recipiente actual con los objetos más grandes que quepan.

La ventaja de esto es que el espacio a priori se aprovecha mejor que con la primera solución puesto que los objetos grandes no desperdician tanto espacio de los recipientes en los que se almacenan. Sin embargo, esto tiene un impacto negativo en la eficiencia que se detallará posteriormente.

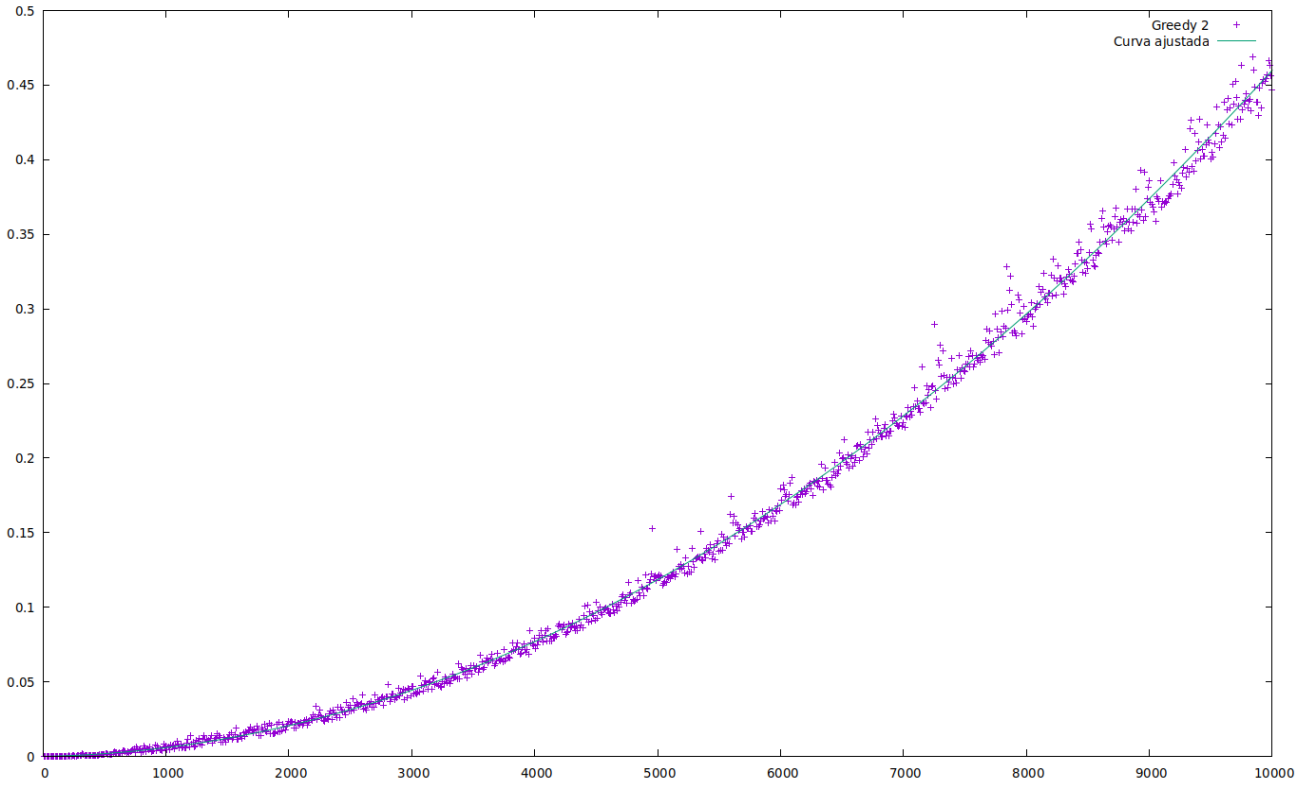
El algoritmo resultante vuelve a tener una complejidad cuadrática ($O(n^2)$, aunque recalcando de nuevo que las constantes serán peores).

*Hacemos uso de la función **compare2** y **qsort** proporcionados por la STL para realizar la ordenación de mayor a menor*

```
int compare2 (const void * a, const void * b){
    return ( *(int*)b - *(int*)a );
}

float funcionSeleccion2(vector<float> & v, float actual,
vector<float>::iterator & posicion){
    for (posicion=v.begin(); posicion != v.end(); ++posicion){
        if (actual + (*posicion) <= R){
            return (*posicion);
        }
    }
    posicion = v.begin();
    return *posicion;
}

vector<pair<float,int>> algoritmoVoraz2(){

    qsort(w, n, sizeof(float), compare2); // Ordenamos de los pesos de
los objetos el vector por quicksort

    vector<float> pesos(w,w+n);       // Vector pesos
    vector<pair<float,int>> solucion; // Vector solución con peso y
recipiente en el que se ha guardado
    float actual;                     //Peso objeto a introducir en el
recipiente
    int n_recipientes = 1;            //Numero de recipientes utilizados
    float recipiente_actual = 0.0;    //Tamaño ocupado del recipiente
```

```
actual
    vector<float>::iterator posicion;

    while (pesos.size() != 0){
        actual = funcionSeleccion2(pesos, recipiente_actual, posicion);
        cout << "Escogido: " << actual << endl;
        pesos.erase(posicion); // Eliminamos el objeto actual que ya
hemos introducido previamente.

        if (actual + recipiente_actual <= R){   //Si cabe en el
recipiente actual, lo insertamos
            solucion.push_back(pair<float,int>(actual,n_recipientes));
            recipiente_actual+=actual;
        }
        else {  //Si no cabe lo insertamos en el siguiente recipiente
            n_recipientes++;
            solucion.push_back(pair<float,int>(actual,n_recipientes));
            recipiente_actual = actual;
        }
    }

    return solucion;
}
```

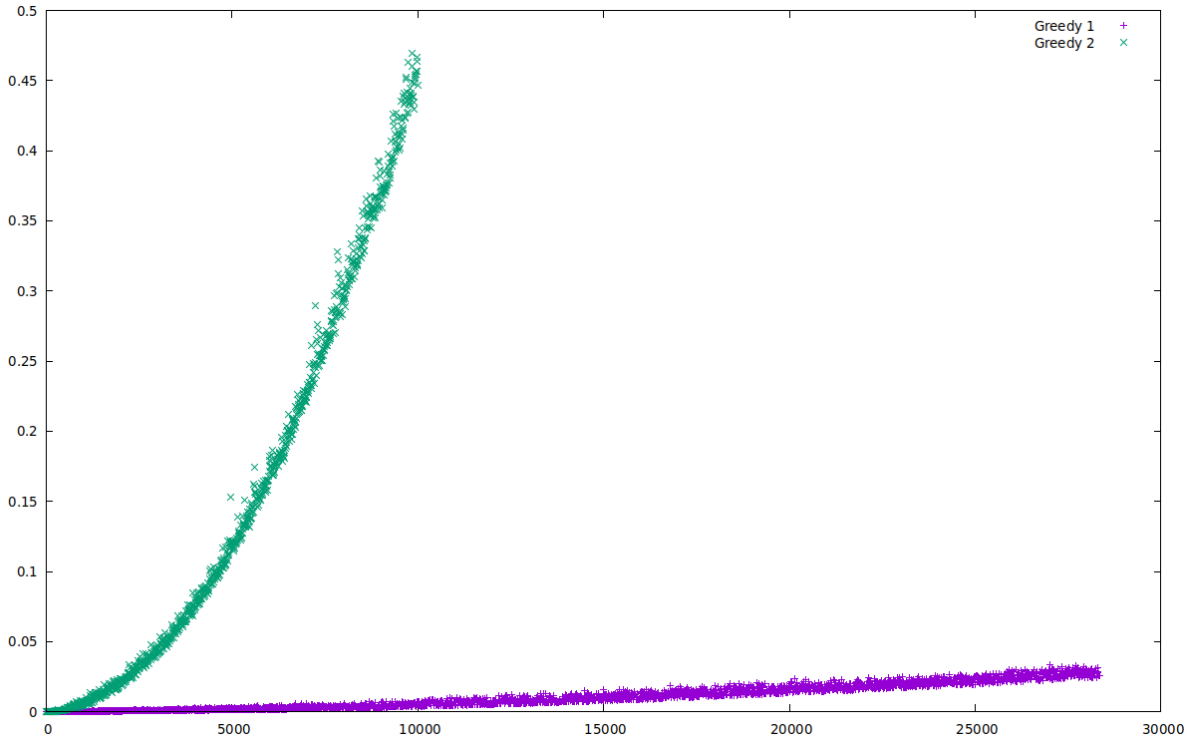
Gráfica:



Constantes ocultas

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
d               = 4.46072e-09      +/- 2.741e-11    (0.6145%)
e               = 1.38397e-06      +/- 2.834e-07    (20.47%)
f               = 9.01761e-05      +/- 0.0006142    (681.1%)

correlation matrix of the fit parameters:
                d      e      f
d               1.000
e              -0.968  1.000
f               0.746 -0.867  1.000
```

Comparación de eficiencia:



En la gŕafica se puede ver que es mucho más eficiente la primera solución que la segunda. Esto se debe a que en la segunda es necesario realizar una búsqueda en los objetos para llegar a los tamaños que caben en el recipiente actual, lo que tiene bastante impacto en las constantes de la función. No obstante, puesto que el orden de eficiencia es el mismo, para tamaños grandes se acercarían sustancialmente más.

Vemos cómo, a pesar de que los dos algoritmos sean del orden $O(n^2)$, las constantes ocultas juegan un papel muy importante en los tiempos de ejecución.

Comparación de optimalidad:

Para realizar la comparación de la optimalidad hemos comparado primero nuestros resultados con el algoritmo de fuerza bruta que se nos proporciona.

```
Los pesos son:
 0.871845  0.783319  0.297428  0.731423  0.166111  0.759542  0.107488  0.69598  0.573772  0.0891034  0.936324  0.52613
tiempo: 29.4215
Se usan 8 recipientes
La distribucion es:
 0.871845  en recipiente 1
 0.783319  en recipiente 2
 0.297428  en recipiente 3
 0.731423  en recipiente 4
 0.166111  en recipiente 2
 0.759542  en recipiente 5
 0.107488  en recipiente 1
 0.69598  en recipiente 3
 0.573772  en recipiente 6
 0.0891034  en recipiente 4
 0.936324  en recipiente 7
 0.52613  en recipiente 8
```

De aquí obtenemos que la solución óptima usa 8 recipientes para esos 12 objetos.
Si probamos nuestros algoritmos con los mismos objetos de entrada obtenemos lo siguiente:

Algoritmo greedy 1

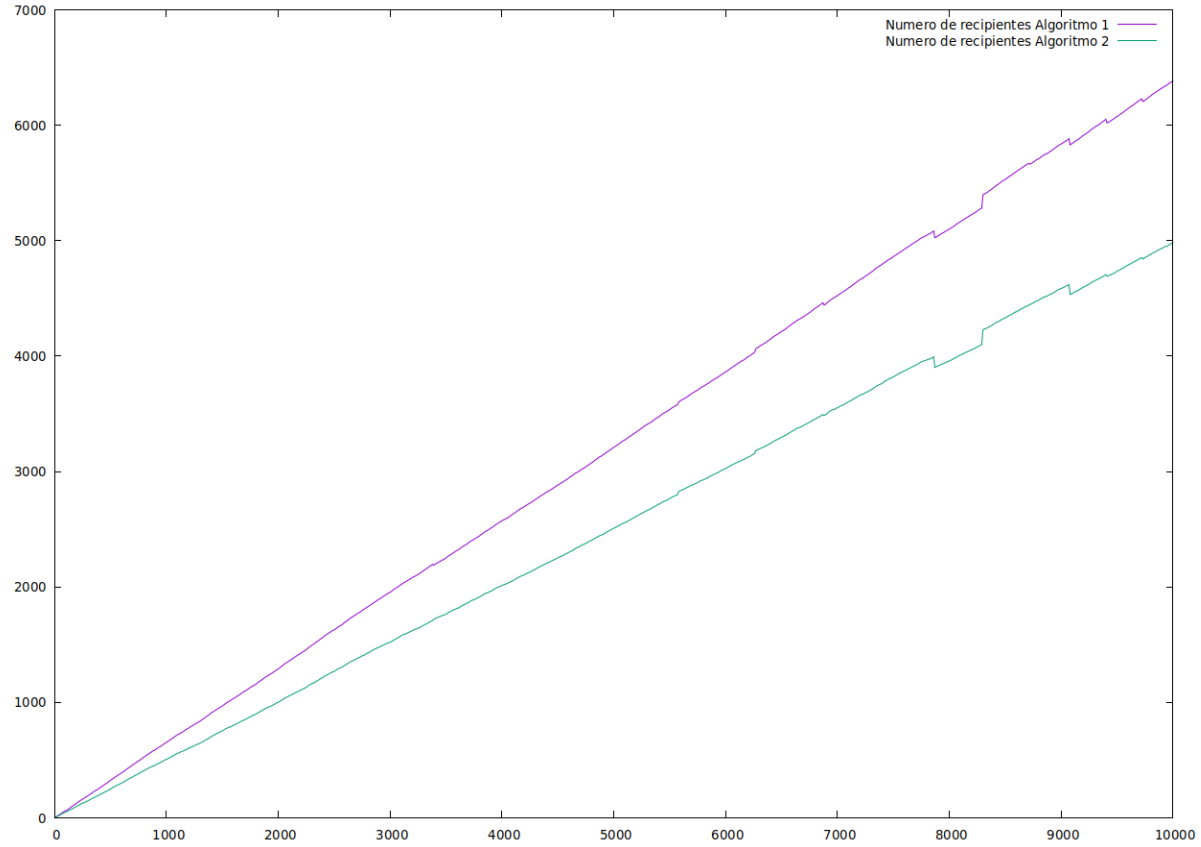
```
Los pesos son:
 0.871845  0.783319  0.297428  0.731423  0.166111  0.759542  0.107488  0.69598  0.573772  0.0891034  0.936324  0.52613

tiempo: 1.2e-05
El tamaño de los recipientes (R) es: 1

Se usan 9 recipientes

La distribucion es:
 0.0891034 en recipiente 1
 0.107488 en recipiente 1
 0.166111 en recipiente 1
 0.297428 en recipiente 1
 0.52613 en recipiente 2
 0.573772 en recipiente 3
 0.69598 en recipiente 4
 0.731423 en recipiente 5
 0.759542 en recipiente 6
 0.783319 en recipiente 7
 0.871845 en recipiente 8
 0.936324 en recipiente 9
```

Algoritmo greedy 2:

```
Los pesos son:
 0.871845  0.783319  0.297428  0.731423  0.166111  0.759542  0.107488  0.69598  0.573772  0.0891034  0.936324  0.52613

tiempo: 2e-05
El tamaño de los recipientes (R) es: 1

Se usan 8 recipientes

La distribucion es:
 0.936324 en recipiente 1
 0.871845 en recipiente 2
 0.107488 en recipiente 2
 0.783319 en recipiente 3
 0.166111 en recipiente 3
 0.759542 en recipiente 4
 0.0891034 en recipiente 4
 0.731423 en recipiente 5
 0.69598 en recipiente 6
 0.297428 en recipiente 6
 0.573772 en recipiente 7
 0.52613 en recipiente 8
```

Vemos como en este caso, el algoritmo más lento obtiene una solución óptima, mientras que usando el algoritmo 1, el más rápido, no logramos una solución óptima (obtenemos 9 recipientes en lugar de 8).

Además, esta diferencia se vuelve más notable a medida que aumentamos el número de objetos.



Vemos cómo el algoritmo 1 obtiene un número de recipientes mayor al del algoritmo 2 conforme aumentan los objetos.

A modo de conclusión, con los resultados obtenidos únicamente podemos asegurar que la solución 1 no siempre obtiene los resultados óptimos, ya que la solución 2 sí que la ha obtenido en el ejemplo, pero eso no afirma que la obtenga para tamaños mayores o simplemente otros objetos diferentes. Sin embargo, debido a la similitud con el problema de la mochila binaria, creemos con convicción que los algoritmos greedy no son un buen diseño para este problema ya que no son capaces de obtener una solución óptima en algunos casos.

Porcentajes de participación:

Manuel Contreras Orge: 20%
Raúl Durán Racero: 20%
Alberto Palomo Campos: 20%
Abel Rios González: 20%
Salvador Romero Cortés: 20%