PRÁCTICA 3 - Algoritmos Greedy

Algorítmica - GII 2ºA1_1

1. Adquisición diferida

Tras analizar el problema e intentar alcanzar una solución óptima, hemos llegado a la siguiente conclusión:

- Se define el factor de incremento: $r_i > 1$ y el precio de $A_i = 1000 \cdot (r_i)^t$, siendo *t* el número de meses que han pasado hasta comprar el equipo.
- El algoritmo se encargará de escoger los equipos de **mayor incremento mensual**, es decir, los escogerá de orden decreciente respecto al factor r_i. De esta forma, el primer mes se compraría el de mayor incremento mensual, de manera que no continuasen pasando los meses y, por consiguiente, el equipo encareciendo considerablemente su valor. Habrá otros cuyo factor de incremento sea inferior y, tras el transcurso de *t* meses seguirán siendo más baratos que los de mayor incremento.

  Ejemplo:

Equipo	Precio	r_i	Precio mes 1	Precio mes 2	Precio mes 3
A	1000	$r_a = 0,6$	1600	2560	4096
B	1000	$r_b = 0,3$	1300	1690	2197

  Se puede comprobar cómo vale más la pena ir comprando los de mayor incremento mensual y ahorrar en cuestión del paso de los meses.

Llegados a este punto, los elementos del algoritmo de Greedy a construir son:

★ Conjunto de candidatos (C): conjunto de todos los equipos (cada equipo está representado por su incremento).

★ Conjunto de seleccionados (S): Equipos seleccionados.

★ Función solución (seHanCompradoTodosLosEquipos): la solución se obtiene una vez se han comprado todos los equipos.

★ Función de factibilidad: todos los candidatos seleccionados son factibles, por lo que no hace falta función de factibilidad.

★ Función selección (seleccionarEquipoMaxIncremento): el candidato a seleccionar en cada momento es el que tenga mayor incremento.

★ Función solución: devuelve el dinero total gastado al transcurrir los *t* meses.

Pseudocódigo del algoritmo

```
S=Ø;
precio_total = 0;
num_meses_pasados = 0;

mientras ! seHanCompradoTodosLosEquipos()
    x = seleccionarEquipoMaxIncremento(C);
    C.eliminar(x);
    precio = calcularPrecio (x, num_meses_pasados)
    precio_total += precio;
    S.añadir(x);
    num_meses_pasados++;
fin
Devolver total;
```

La función calcular precio realizaría la siguiente operación:

$$precio = 1000(C[i])^{num\_meses\_pasados}$$

Demostración de optimalidad

Sea $(x_1, x_2,...,x_n)$ los equipos seleccionados por nuestro algoritmo con un precio total:

$$\text{Coste} = \sum_{t=0;\, i=0}^{n} 1000(r_i)^t = 1000 \sum_{t=0;\, i=0}^{n} (r_i)^t = 1000 \sum_{t=0}^{n} (r_t)^t$$

Supongamos que hay otra ordenación intercambiando dos elementos x_i y x_j cuyo coste total es menor:

$$1000\left(\sum_{t=0}^{i-1} r_t^t + r_i^i + r_j^j + \sum_{t=j+1}^{n} r_t^t\right) > 1000\left(\sum_{t=0}^{i-1} r_t^t + r_j^i + r_i^j + \sum_{t=j+1}^{n} r_t^t\right)$$

$$\left(\sum_{t=0}^{i-1} r_t^t + r_j^i + r_i^j + \sum_{t=j+1}^{n} r_t^t\right) > \left(\sum_{t=0}^{i-1} r_t^t + r_j^i + r_i^j + \sum_{t=j+1}^{n} r_t^t\right)$$

$$r_i^i + r_j^j > r_j^i + r_i^j$$

De los que obtenemos $r_i > r_j$ y $i < j$, pero por la naturaleza de nuestro algoritmo, éste hubiera seleccionado r_i antes que r_j => **<u>contradicción</u>**.

2. El problema de los recipientes

A continuación, hemos implementado dos algoritmos voraces enfocados a la resolución del problema de los recipientes.

Los elementos del algoritmo de Greedy a construir son:

★ Conjunto de candidatos (C): Todos los objetos a meter en los recipientes.

★ Conjunto de seleccionados (S): Objetos en el orden en el que se van seleccionando.

★ Función solución (seHanSeleccionadoTodosLosObjetos): El algoritmo finaliza tras seleccionar todos los objetos.

★ Función de factibilidad: todos los candidatos seleccionados son factibles, por lo que no hace falta función de factibilidad, ya que el número de recipientes es ilimitado.

★ Función selección: definimos dos funciones de selección:

1. SeleccionaMayorPesoQueCabe(): Selecciona el objeto de mayor peso que cabe en el recipiente actual, devolviendo su posición, si ninguno cabe devuelve -1.

2. SeleccionarSiguienteObjetoQueCabe(): Selecciona el primer objeto del conjunto de candidatos que cabe en el recipiente actual devolviendo su posición, es decir, comprueba si el primer elemento cabe, si no pasa al siguiente y así sucesivamente. Si ninguno cabe devuelve -1.

★ Función solución: devuelve el número total de recipientes usados al seleccionar todos los objetos.

Pseudocódigo del algoritmo

```
S=Ø;
recipientes_usados = 1;
restante_actual = 1;

mientras quedanObjetos()
    pos = FuncionSeleccion(C);
    si (pos != -1)
        restante_actual -= C[pos];
        S.añadir(pos);
        C.eliminar(pos);
    sino
        restante_actual = 1;
        recipientes_usados++;
    fin
fin

Devolver recipientes_usados;
```

Implementación

Función greedy:

```
int funcionGreedy(){

    int recipientes = 1, posObjeto;
    double restante_actual = 1;

    while( !C.empty() ){
        posObjeto = funcionSeleccion(restante_actual);

        if(posObjeto != -1){
            restante_actual -= C[posObjeto];
            S.push_back(C[posObjeto]);
            C.erase(C.begin() + posObjeto);
        }
        else{
            recipientes++;
            restante_actual = 1;
        }
    }

    return recipientes;
}
```

Funciones Selección:

1. SeleccionaMayorPesoQueCabe()

```
int seleccionarMayorPesoQueCabe(double pesoRestante){
    //En el caso de que no haya un objeto que quepa
    int pos = -1;
    double maximoPeso = 0;
    for(int i = 0; i < C.size(); i++)
        if(C[i] <= pesoRestante && C[i] >= maximoPeso){
            pos = i;
            maximoPeso = C[i];
        }
        /* Si hay un objeto que cabe y tiene mayor peso del que
        se habia guardado, se reemplaza. */
    return pos;
}
```

2. SeleccionarSiguienteObjetoQueCabe()

```
int seleccionarSiguiente(double pesoRestante){
    bool cabe = false;
    int pos = -1;

    for (int i = 0; i < C.size() && !cabe; i++)
        if (C[i] <= pesoRestante ){
            pos = i;
            cabe = true;
        }
    return pos;
}
```

Comparación ejecución fuerza bruta y greedy

Ejecución fuerza bruta para tamaño 11:

```
$ ./recipientes-fuerzabruta 11
  Los pesos son:
  0.866929  0.471104  0.847393  0.134526  0.978849  0.51658
  0.161067  0.0484861  0.905923  0.847063  0.59071
  tiempo: 1.75205
  Se usan 7 recipientes
  La distribucion es:
   0.866929  en recipiente 1
   0.471104  en recipiente 2
   0.847393  en recipiente 3
   0.134526  en recipiente 3
   0.978849  en recipiente 4
   0.51658   en recipiente 2
   0.161067  en recipiente 5
   0.0484861 en recipiente 1
   0.905923  en recipiente 6
   0.847063  en recipiente 7
   0.59071   en recipiente 5
```

Ejecución para algoritmo greedy con función de selección que selecciona el mayor peso en cada iteración.

```
$ ./recipientes-greedy1 11
  Vector inicial:
  0.866929 0.471104 0.847393 0.134526 0.978849 0.51658 0.161067
  0.0484861 0.905923 0.847063 0.59071
  10 1.5e-05
  Recipientes: 7
  La distribucion es:
   0.978849  en recipiente 1
   0.905923  en recipiente 2
   0.0484861 en recipiente 2
   0.866929  en recipiente 3
   0.847393  en recipiente 4
   0.134526  en recipiente 4
   0.847063  en recipiente 5
   0.59071   en recipiente 6
   0.161067  en recipiente 6
   0.51658   en recipiente 7
   0.471104  en recipiente 7
```

Ejecución para algoritmo greedy con función de selección que selecciona el siguiente objeto que cabe:

```
$ ./recipientes-greedy2 11

  Vector inicial:
  0.866929 0.471104 0.847393 0.134526 0.978849 0.51658 0.161067
  0.0484861 0.905923 0.847063 0.59071

  10 1.5e-05

  Recipientes: 8

  La distribucion es:
   0.866929  en recipiente 1
   0.0484861 en recipiente 1
   0.471104  en recipiente 2
   0.134526  en recipiente 2
   0.161067  en recipiente 2
   0.847393  en recipiente 3
   0.978849  en recipiente 4
   0.51658   en recipiente 5
   0.905923  en recipiente 6
   0.847063  en recipiente 7
   0.59071   en recipiente 8
```

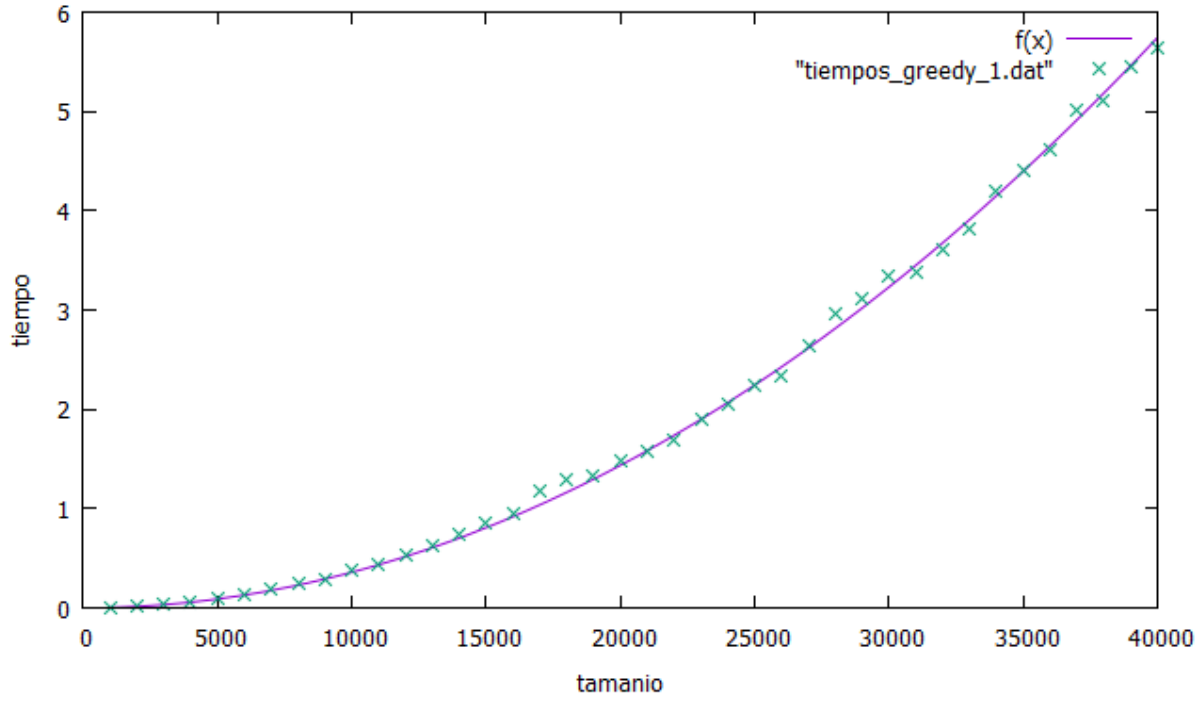
Tras varias ejecuciones, hemos podido observar que en el algoritmo que selecciona el de más peso devuelve la solución óptima muchas más veces y tiene menor margen de error que el algoritmo que selecciona el siguiente objeto que cabe, ya que de la manera en la que está implementado, se acercará mucho más a la solución que da el algoritmo fuerza bruta.

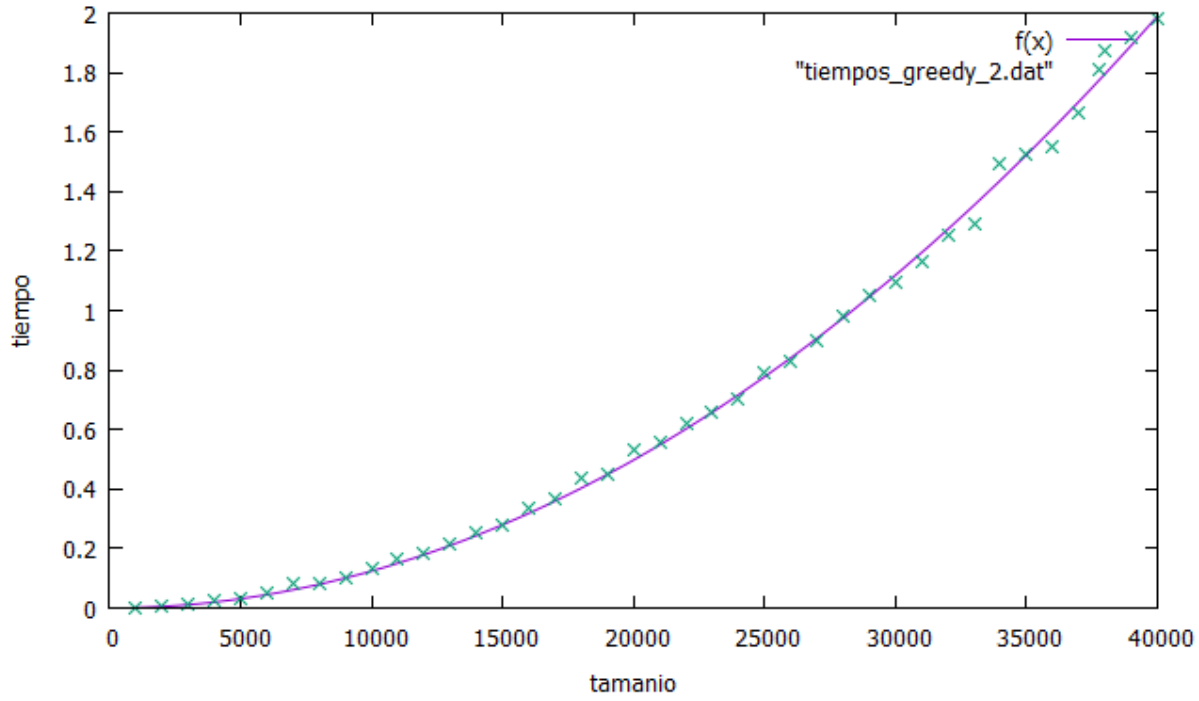
Y aunque el algoritmo que selecciona el siguiente objeto que cabe en el recipiente es algo más rápido ya que realiza menos iteraciones, la solución que se obtiene en la mayoría de los casos es peor que para la otra función de selección.

Estudio de la Eficiencia

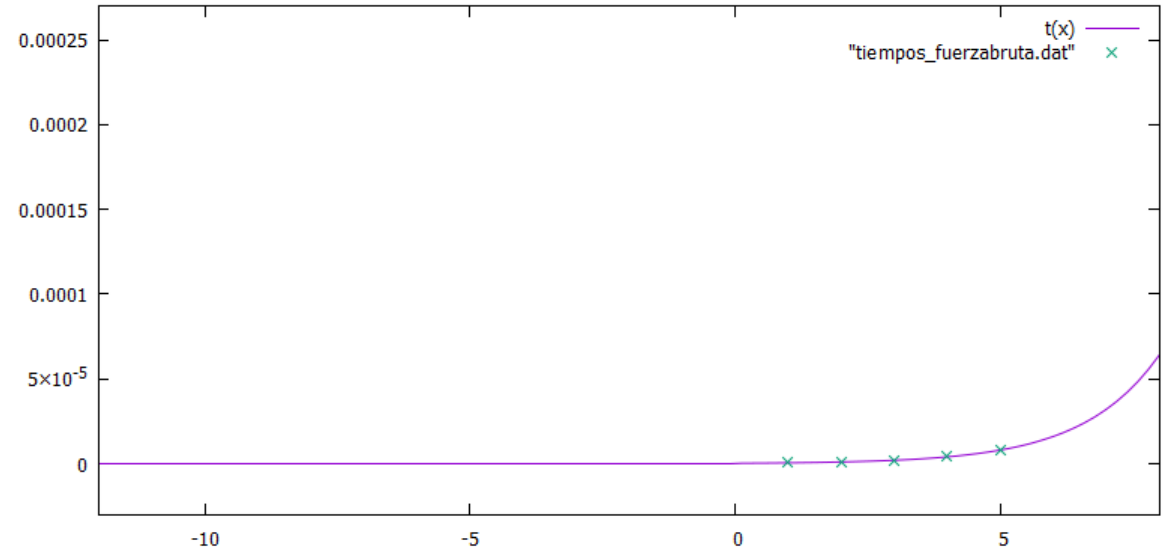
Hemos realizado un pequeño estudio de la eficiencia de los algoritmos que hemos implementado.

Eficiencia Híbrida





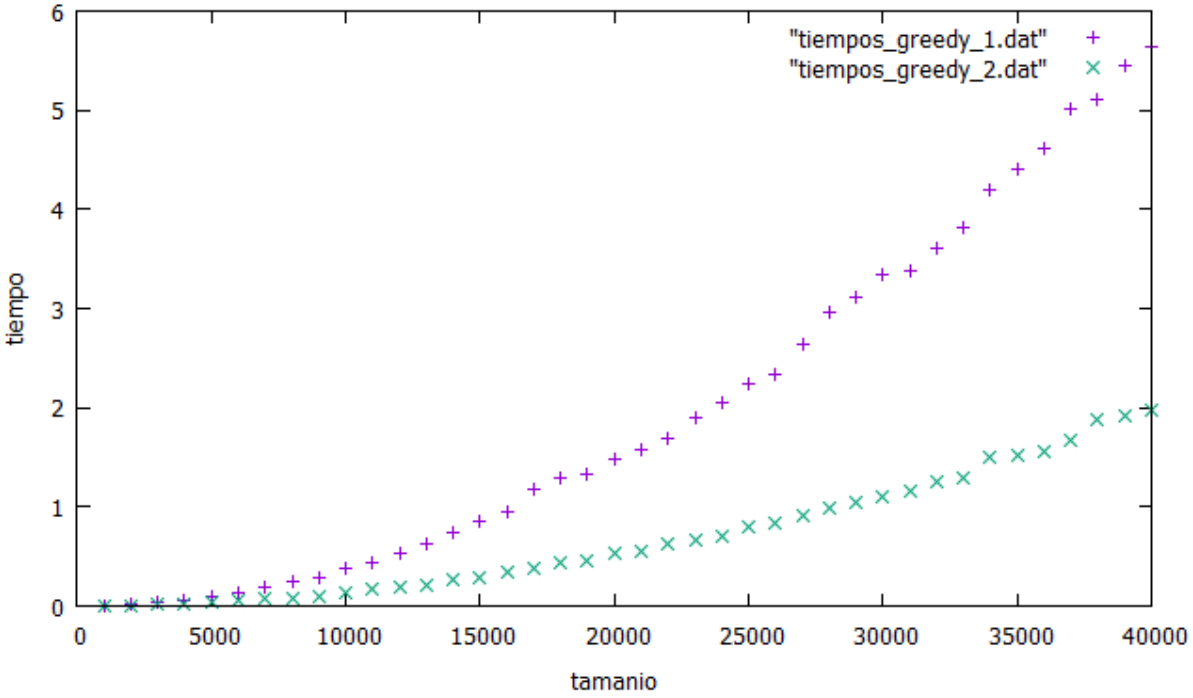
Como podemos ver, los algoritmos Greedy se ajustan perfectamente a n^2.



Con el algoritmo de fuerza bruta observamos que se ajusta bastante bien a un algoritmo tipo 2^x.

Eficiencia empírica Greedy

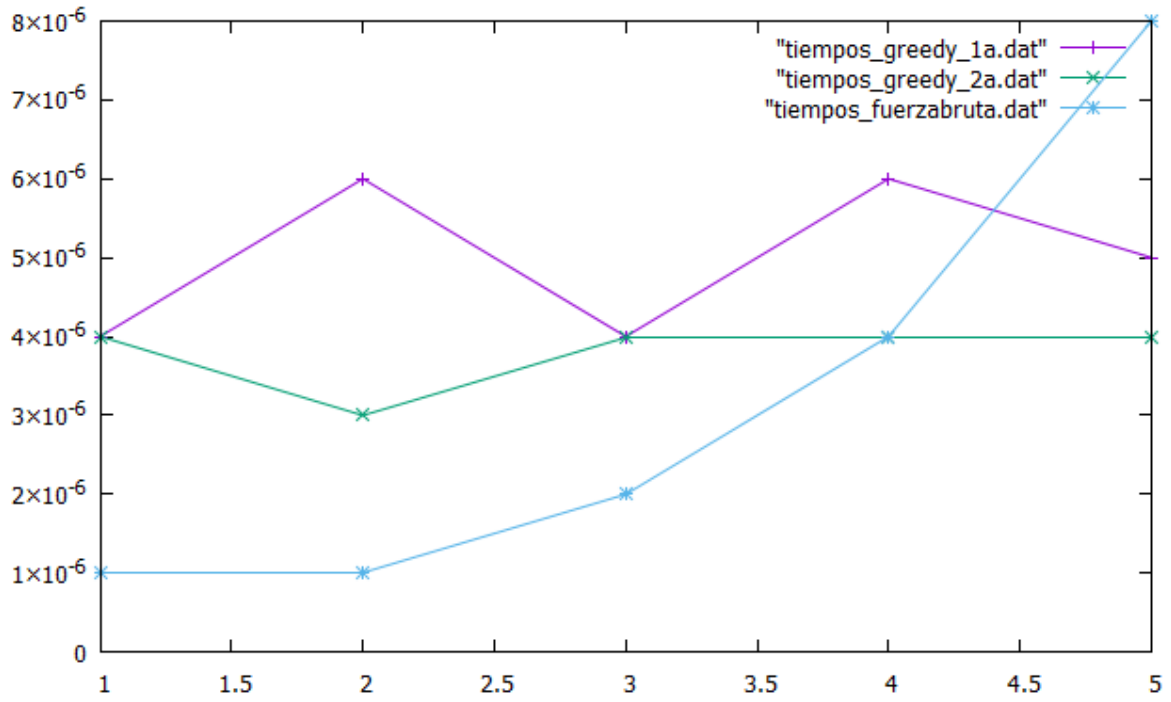
Después la comparación que vamos a hacer es la de los dos algoritmos Greedy, sin tener en cuenta el algoritmo de fuerza bruta:



Podemos observar que el algoritmo algoritmo que selecciona el de más peso (greedy 1) es menos eficiente que el que va seleccionando según van cabiendo (greedy 2), pero ya hemos visto antes que aunque sea más eficiente el segundo, la solución que devuelve en muchos casos es más lejana a la óptima.

Eficiencia empírica Fuerza bruta y Greedy

Ahora veremos la comparación entre los 3 algoritmos, Greedy 1, Greedy 2 y fuerza bruta:



Podemos visualizar un fenómeno bastante curioso. Lo primero es que solamente hemos comprobado los tiempos con 5 iteraciones desde 1 hasta 5 con incremento 1, ya que con más tamaño el algoritmo fuerza bruta empieza a incrementar el tiempo que tarda de forma exponencial. Por eso, resulta curioso el hecho que de primeras es más eficiente que los otros 2 algoritmos, pero se observa claramente cómo empieza a incrementar el tiempo rápidamente conforme aumenta el tamaño.

Porcentajes

- Marina Muñoz Cano: 20%
- Mario López González: 20%
- Patricia Cabrera Marrero: 20%
- Pablo Borrego Megías: 20%
- Antonio Quirante Hernández: 20%