DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

TP1 : Problema de la mochila

Dame la mochila!

Algoritmos y Estructura de Datos III

Integrante	LU	Correo electrónico
Destuet, Carolina	753/12	carolinadestuet@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

I Introducción teórica del problema **3**

1. Descripción **3**

II Resolución **4**

2. Resolución propuesta 1 : Fuerza Bruta **4**
2.1. Pseudocódigo y complejidades 4

3. Resolución propuesta 2: Backtracking con podas **5**
3.1. Poda por factibilidad 5
3.1.1. Pseudocódigo y complejidades 5

4. Resolución propuesta 3: Programación Dinámica **6**
4.1. Pseudocódigo y complejidades 8

III Experimentación **9**

5. Fuerza Bruta **9**

6. BT con poda **9**

7. Programación Dinámica **11**

IV Discusión y conclusiones **14**

8. Análisis comparativo entre algoritmos **14**

Parte I
Introducción teórica del problema

1. Descripción

El "Problema de la mochila", también conocido en inglés como "Knapsack Problem", se lo puede describir formalmente de la siguiente manera :

Dado un conjunto finito de objetos $T = \{o_1, ..., o_n\}$, en donde para cada i el objeto o_i tiene un costo c_i y un beneficio b_i asociados, junto con un valor de restricción V, se quiere hallar un subconjunto de objetos $S \subset T$ tal que

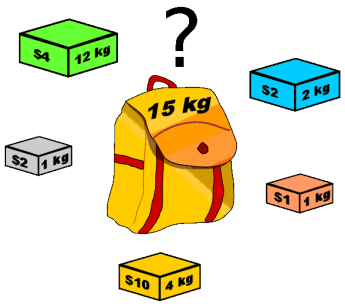
$$\sum_{o_i \in S} c_i \leq V$$

es decir, tal que la suma de sus costos no exceda el valor de restricción, y tal que

$$\forall S', S' \subset T \wedge \sum_{o_i \in S'} c_i \leq V \implies \sum_{o_i \in S'} b_i \leq \sum_{o_i \in S} b_i$$

o sea, tal que la suma de sus beneficios sea máxima en el sentido de que para cualquier otro subconjunto S' que cumpla con el valor de restricción (la suma de los costos no supera al valor restrictivo), el beneficio total (la suma de los beneficios) de S' no sea mayor al beneficio total de S.

Una representación típica del problema descripto muy ilustrativa y concreta es suponer que se tiene una mochila con cierta capacidad fija de peso que puede soportar y, por otro lado, se dispone de n objetos que nos interesa llevar en la mochila (una botella de agua, un libro, etcétera). Cada objeto tiene asociado dos atributos : un peso de carga y un beneficio que denota el nivel de importancia o valor que se le da a este objeto. El problema de interés es entonces poder descifrar cómo cargar la mochila tomando alguna combinación de los n objetos tal que esa combinación no supere la capacidad que pueda soportar aquella, y tal que dé el máximo valor de beneficio entre todos los subconjuntos de items que no superen el peso de carga. Notar por un lado, que puede existir más de una combinación de objetos posible que cumpla con lo pedido, y, por otro lado, que dependiendo de los valores asociados al peso de la mochila y al de los objetos, es posible que no entren el total de los n objetos en la mochila. En este último caso es en donde es de gran utilidad y no es trivial poder resolver el problema planteado puesto que hay que decidir efectivamente qué objetos se pueden tomar y cuáles descartar.



Parte II
Resolución

A continuación se da una breve descripción de los algoritmos propuestos para la resolución del problema de la mochila junto con el pseudocódigo y justificación de la complejidad de cada uno de ellos. Para los algoritmos en conjunto debe tenerse presente que los parámetros de entrada son una instancia de clase mochila M, que tiene como miembros un entero W, que representa el peso de la mochila, y un vector v de pares de enteros, donde $v[i]$ es $< beneficio, peso >$ del iésimo ítem, además llamaremos n al número total de objetos a considerar.

2. Resolución propuesta 1 : Fuerza Bruta

La primera resolución se trata de un algoritmo de fuerza bruta, esta es una técnica muy directa para resolver problemas : se enumeran todas los posibles candidatos a solución, sin descartar nada a priori y se chequea si cada candidato cumple con las condiciones del enunciado. En este contexto, si hay n objetos para elegir, entonces se tienen 2^n posibles combinaciones de objetos para meter en la mochila. Tanto este algoritmo como los algoritmos de backtracking con distintas podas propuestos están basados en el método recursivo de backtracking. Este consiste en la construcción implícita de un árbol binario de soluciones. Cada nivel del árbol representa una elección que depende del nivel inmediato superior, y cada solución posible está representada por un camino que empieza en la raíz y que termina en una hoja. La raíz es el nivel cero y significa un estado en el que no se construyó ninguna solución parcial todavía. Una hoja representa el estado en donde todas las decisiones que dan lugar a una solución fueron tomadas. En el contexto del problema de la mochila, el nivel k representa el estado en el que se ha decidido cuáles de los primeros k elementos fueron o no incluidos en la mochila. Hay 2^k nodos en este nivel y las hojas del árbol (o sea, las posibles soluciones) están en el nivel n.

2.1. Pseudocódigo y complejidades

FuerzaBrutaRecur(Mochila M, int paso, int pesoActual, int benefActual, int benefMax, int contador)

if $paso == tam(M.items)$ **then**
  if $(pesoActual <= M.capacidad)$**and**$(benefAct > benefMax)$ **then**
    $benefMax \Leftarrow benefAct$
  end if
else
  $contador = contador + 2$
  $FuerzaBrutaRecur(paso + 1, pesoActual, benefActual, benefMax, contador)$
  $pesoActualizado \Leftarrow pesoActual + peso(M.items[paso])$
  $benefActualizado \Leftarrow benefActual + beneficio(M.items[paso])$
  $FuerzaBrutaRecur(paso + 1, pesoActualizado, benefActualizado, benefMax, contador)$
end if

Complejidad: $\mathcal{O}(2^n)$ Como se puede ver en el pseudocódigo, nuestro algoritmo tiene un condicional ïf-else"global donde en la parte del ïf"hay un número constante de asignaciones y comparaciones entre variables enteras, por lo que todo lleva un costo total $\mathcal{O}(1)$; por otro lado, la parte del .else"tiene un número constante de asignaciones y comparaciones de variables enteras y tiene dos llamadas recursivas a nuestra función incrementando en 1 al parámetro paso, lo cual es análogo a decrementar en 1 a n, con n la cantidad de ítems. Por lo tanto, nuestra función recursiva tiene como ecuación de recurrencia :

$$T(n) = 2T(n-1) + \mathcal{O}(1)$$

Vamos a probar por inducción que la complejidad asociada a esta recurrencia es $\mathcal{O}(2^n)$:

- Caso Base :$n = 0$ Como para el problema de la mochila esta función se inicializa con parámetros $FuerzaBrutaRecur(M, 0, 0, 0, 0, 0)$, luego, se evalúa la condición del if $paso == tam(M.items)$ como verdadera y solamente hay una comparación y una asignación de variables enteras, por lo

que la complejidad de "FuerzaBrutaRecur"para este caso es $\mathcal{O}(1)$, y como $1 = 2^0$, se cumple la hipótesis inductiva para el caso base.

- Hipótesis inductiva : $P(n) \implies P(n+1)$ Si $P(k)$ es la propiedad "La función FuerzaBrutaRecur tiene complejidad $\mathcal{O}(2^k)$"siendo k la cantidad de ítems del problema de la mochila, quiero ver que asumiendo que vale $P(n)$, entonces vale $P(n+1)$. Si volvemos a la ecuación de recurrencia, tenemos

$$T(n+1) = 2T(n) + \mathcal{O}(1) =_{HI} 2 * 2^n + \mathcal{O}(1) = \mathcal{O}(2^{n+1})$$

3. Resolución propuesta 2: Backtracking con podas

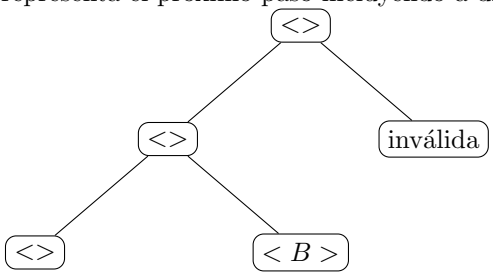
3.1. Poda por factibilidad

Por cómo está caracterizado nuestro problema de interés, podemos testear a partir de un nodo interno cuan factible es llegar a una solución completa desde la solución parcial construida hasta ese nodo. Por ejemplo, si hemos construido una solución parcial con los elementos $\{x_{i_1}, ..., x_{i_m}\}$ con $m < n$ y estamos parados en un nodo con el elemento $x_{i_{m+1}}$ teniendo

$$(\sum_{j=1}^{m} peso(x_{i_j})) + peso(x_{i_{m+1}}) > W$$

.

entonces no necesitamos seguir contemplando soluciones extendiendo a la solución parcial $< x_{i_1}, ..., x_{i_{m+1}} >$ dado que no hay una solución factible que se construya a partir de ella puesto que incumpliría la restricción impuesta con W (por supuesto que estamos teniendo en cuenta que el peso asociado a un ítem no puede tomar valores negativos). Hecha esta observación, podamos el árbol a partir de esta rama, sin considerar los descendientes a partir del elemento $x_{i_{m+1}}$ y la solución parcial con los elementos $x_{i_1}, ..., x_{i_m}$.
A modo de ilustración, damos un gráfico del árbol para el caso $W = 2$ y el conjunto de ítems $\{A, B\}$ con $A =< beneficio = 4, peso = 4 >$, $B =< beneficio = 3, peso = 2 >$. Parados en un nodo en el nivel i, el hijo izquierdo de i representa el próximo paso de la solución parcial sin tomar el ítem iésimo y el hijo derecho representa el próximo paso incluyendo a dicho ítem en la solución.



En nuestro algoritmo dado, la solución al problema se guarda en una variable "benefMax", que se guarda el mejor beneficio máximo encontrado hasta el momento, si al llegar a una hoja del árbol, el beneficio de la solución que comprende de la raíz a dicha hoja es mayor al valor que hay en "benefMax", entonces "benefMax"se actualiza siendo el beneficio total de esta solución el valor asignado a "benefMax".

3.1.1. Pseudocódigo y complejidades

Complejidad : $\mathcal{O}(2^n)$

Según se puede ver en el pseudocódigo, nuestro algoritmo tiene un condicional ïf-else"global donde en la parte del ïf"hay un número constante de asignaciones y comparaciones entre variables enteras, por lo que todo lleva un costo total $\mathcal{O}(1)$; por otro lado, la parte del .else"tiene otro condicional anidado del tipo ïf-else", en donde en ïf"tenemos número constante de asignaciones y comparaciones de variables enteras y tenemos dos llamadas recursivas a nuestra función incrementando en 1 al parámetro paso, lo cual es análogo a decrementar en 1 a n, con n la cantidad de ítems ; mientras que en el .else"solamente hay una llamada recursiva a la función. Por lo tanto, acotando por el peor caso del condicional, el ïf", nuestra función recursiva tiene como ecuación de recurrencia :

BTpodaFactRecur(Mochila M,int paso, int pesoActual, int benefActual, int benefMax, int contador)

if $paso == tam(M.items)$ **then**
  if $benefAct > benefMax$ **then**
    $benefMax \Leftarrow benefAct$
  end if
else
  if $pesoActual + peso(M.items[paso]) \leq capacidad(M)$ **then**
    $contador = contador + 2$
    $BTpodaFactRecur(paso + 1, pesoActual, benefActual, benefMax, contador)$
    $pesoActualizado \Leftarrow pesoActual + peso(M.items[paso])$
    $benefActualizado \Leftarrow benefActual + beneficio(M.items[paso])$
    $BTpodaFactRecur(paso + 1, pesoActualizado, benefActualizado, benefMax, contador)$
  else
    $contador \Leftarrow contador + 1$
    $BTpodaFactRecur(paso + 1, pesoActual, benefActual, benefMax, contador)$
  end if
end if

$$T(n) = 2T(n-1) + \mathcal{O}(1)$$

Se prueba por inducción que la complejidad asociada a esta ecuación de recurrencia es $\mathcal{O}(2^n)$:

- Caso Base :$n = 0$ En tal caso, como para el problema de la mochila esta función se inicializa con parámetros $BTpodaFactRecur(M, 0, 0, 0, 0, 0)$, luego se evalúa la condición del ïf-else"global $paso == tam(M.items)$ como verdadera y solamente hay una comparación y una asignación de variables enteras, por lo que la complejidad de "BTpodaFactRecur"para este caso es $\mathcal{O}(1)$, y como $1 = 2^0$, se cumple la hipótesis inductiva para el caso base.

- Hipótesis inductiva : $P(n) \implies P(n+1)$ Si $P(k)$ es la propiedad "La función BTpodaFactRecur tiene complejidad $\mathcal{O}(2^k)$ siendo k la cantidad de ítems del problema de la mochila, quiero ver que asumiendo que vale $P(n)$, entonces vale $P(n+1)$. Si volvemos a la ecuación de recurrencia, tenemos

$$T(n+1) = 2T(n) + \mathcal{O}(1) =_{HI} 2 * 2^n + \mathcal{O}(1) = \mathcal{O}(2^{n+1})$$

4. Resolución propuesta 3: Programación Dinámica

Muchos problemas pueden ser resueltos separándolos en problemas más chicos, resolviendo estos y combinándolos para obtener una solución al problema inicial. Esta es una técnica clásica conocida como "Divide and Conquer". Ahora bien, si los subproblemas en los que se puede dividir el problema del que partimos no son independientes entre sí, compartiendo subproblemas más chicos, entonces un algoritmo de "Divide and Conquerresuelve los subproblemas comunes repetidas veces. La idea central de Programación Dinámica es evitar realizar trabajo repetido; para ello, se van guardando los resultados ya calculados en alguna estructura con acceso en $\mathcal{O}(1)$ para reusarlos; de alguna forma, estamos cediendo complejidad espacial a cambio de mejorar la complejidad temporal.

Para el diseño del algoritmo de programación dinámica, primero se debe derivar una relación de recurrencia que exprese la solución a una instancia del problema de la mochila en términos de soluciones a instancias más chicas **sin violar el principio de optimalidad**, es decir, la solución óptima del problema global debe incluir la solución óptima de sus subproblemas. Si nos atenemos a este tipo de relaciones de recurrencia, o sea, en las cuales se cumpla el principio de optimalidad, entonces nos aseguramos de que al obtener nuestra solución al problema original a través de soluciones a problemas más chicos, esta será la óptima.

En el algoritmo entonces construimos una tabla T de $n+1$ filas y $W+1$ columnas, para $0 \leq i \leq n$ y $0 \leq p \leq W$, **(*) la entrada $T[i, p]$ guardara el máximo beneficio que se puede obtener de cualquier subconjunto del conjunto de ítems $\{1, ..., i\}$ con la restricción de que el peso**

total de estos no supere el valor p. Si podemos computar todas las entradas de T, entonces la entrada $T[n, W]$ tendrá el beneficio máximo posible de obtener de algún subconjunto de $\{1, ..., n\}$ tal que el peso total de dicho subconjunto no supere el valor W. El algoritmo se define recursivamente de la siguiente forma :
Valores Iniciales:

$$\begin{array}{ll} T[0,p] = 0 & \forall 0 \leq p \leq W \\ T[i,0] = 0 & \forall 0 \leq i \leq n \end{array}$$

Paso recursivo:

$$T[i,p] = \begin{cases} T[i-1,p] & p_i > p \\ max(T[i-1,p], b_i + T[i-1,p-p_i]) & \text{caso contrario} \end{cases}$$

Veamos que este método es correcto, es decir que vale la propiedad (*) mencionada arriba para $T[i, p]$. Para calcular $T[i, p]$ notemos que tenemos dos opciones :
1) Descartar al ítem i : Luego, lo mejor que podemos obtener de beneficio con algún subconjunto de $\{1, ..., i-1\}$ y con capacidad de mochila igual a p es $T[i-1, p]$.
2) Tomar al ítem i (solamente posible si $p_i \leq p$): luego, tenemos un beneficio b_i aportado por el item i pero ahora no nos podemos exceder de un peso de $p - p_i$, lo mejor que podemos hacer con algún subconjunto de $\{1, ..., i-1\}$ y con capacidad de mochila igual a $w - p_i$ es $T[i-1, p-p_i]$. En total obtenemos un beneficio de $b_i + T[i-1, p-p_i]$.
La solución para $T[i, p]$ será entonces $T[i-1, p]$ si no podemos considerar al item i por la restricción $p_i > p$, o, en caso de poder considerarse, será el máximo entre $T[i-1, p-p_i] + b_i$ y $T[i-1, p]$

4.1. Pseudocódigo y complejidades

DynamicProgramming(Mochila M)

```
vector < vector < int >> T ⇐ crearVacio(vector < vector < int >>)
for i = 0 to tam(M.items) do
  vector < int > filaIesima ⇐ crearVacio(vector < int >)
  for j = 0 to M.capacidad do
    filaIesima.push(0)
  end for
  T.push(filaIesima)
end for
for i = 1 to tam(M.items) do
  for p = 0 to M.capacidad do
    if p_i ≤ p then
      if b_i + T[i − 1, p − p_i] > T[i − 1, p] then
        T[i, p] ⇐ b_i + T[i − 1, p − p_i]
      else
        T[i, p] ⇐ T[i − 1, p]
      end if
    else
      T[i, p] ⇐ T[i − 1, p]
    end if
  end for
end for
```

Complejidad: $\mathcal{O}(n * W)$, donde n es la cantidad de ítems y W es el peso de la mochila

Dado que hay dos for globales que iteran $n = tam(M.items)$ y ambos conllevan adentro de su ciclo otro for, el cual itera en ambos casos $W = M.capacidad$ veces, y como además de estos fors anidados, dentro del ciclo de ellos hay un número constante de asignaciones, comparaciones de variables enteras y push de vector que llevan a lo sumo, en el caso del vector, $\mathcal{O}(1)$ amortizado, por lo dicho, la complejidad total del algoritmo es de :

$$2 * (\sum_{i=0}^{n}(\sum_{j=0}^{W} 1)) = 2 * n * \sum_{j=0}^{W} 1 \\ = 2 * n * W$$

Tenemos entonces que efectivamente la complejidad es de $\mathcal{O}(n * W)$

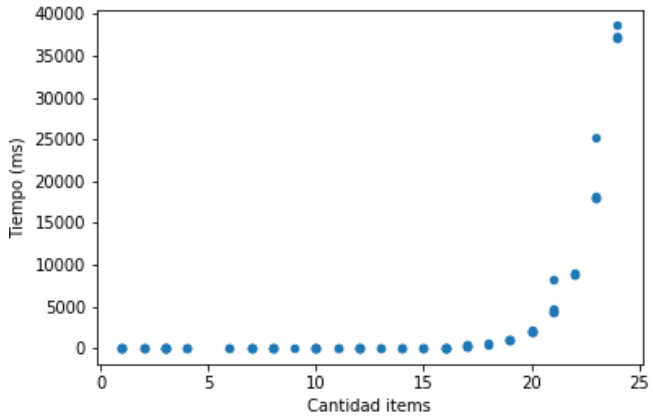
Parte III
Experimentación

5. Fuerza Bruta

Parte de la experimentación relacionada exclusivamente al algoritmo de fuerza bruta fue generar instancias de la mochila con W y n.

Dado que la cota de complejidad planteada teóricamente era $\mathcal{O}(2^n)$ siendo n la cantidad de ítems, se esperaba que el gráfico que muestra la relación entre la cantidad de ítems y el tiempo que toma el algoritmo en resolver el problema fuera exponencial.

El gráfico a continuación expone la relación entre la cantidad de objetos del problema de la mochila y el tiempo que el algoritmo tarda en dar el resultado. Como era de esperar para el algoritmo, el gráfico tiene forma exponencial con una varianza chica de los datos. Casos de tamaño superior a 25 no fueron testeados por el gran costo temporal asociado.

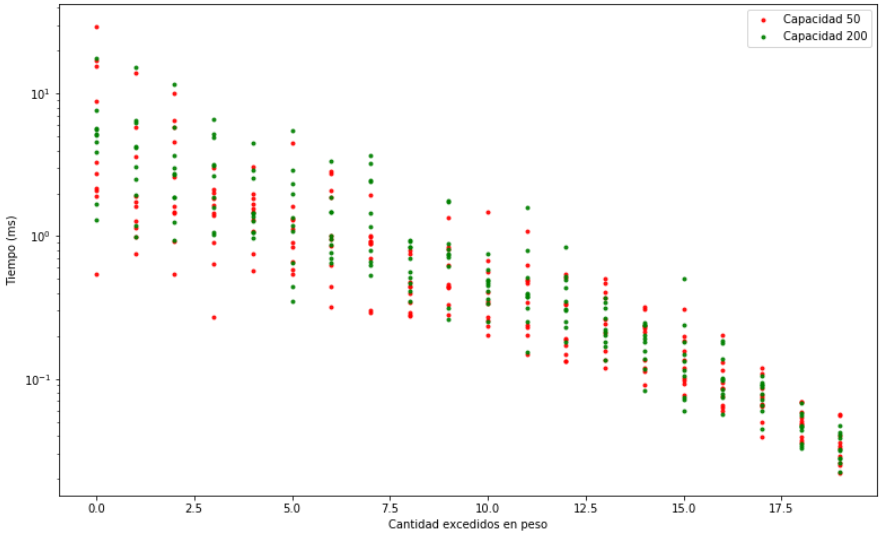


6. BT con poda

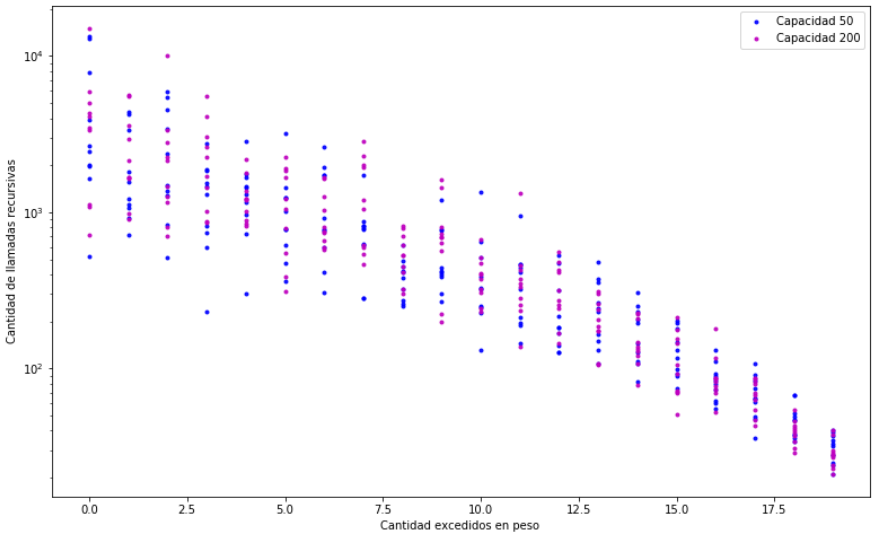
En teoría, en complejidad, Backtracking con poda de factibilidad es igual a Fuerza Bruta. Esto no significa que la poda sea irrelevante, puede disminuir el tiempo de procesamiento significativamente. Esto sucede porque se poda el árbol de soluciones posibles que se analizan. En efecto, en vez de revisar todas las ramas sin excepción, al aplicar podas descartamos caminos que ya sabemos que no nos llevarán a un resultado de interés. Para contrastar empíricamente esto se hizo un test : para $W = 50$ y $W = 200$ se generaron 200 instancias siempre de tamaño $n = 19$, en donde se iba aumentando la cantidad de ítems que superan la capacidad de la mochila sin un orden preciso.

A continuación mostramos dos gráficos relacionados al test. El primero muestra el tiempo de cómputo del algoritmo en función de la cantidad de elementos que tienen un peso mayor a W (para $W = 50$ y $W = 200$). Se puede ver cómo va mejorando el tiempo de procesamiento a medida que se aumentan los ítems a descartar (Çant descartados", o sea, los que exceden el peso de la mochila), que es lo que se había presumido. El segundo gráfico muestra la cantidad de llamadas recursivas (la cual es inversamente proporcional a la cantidad de podas) que hace el algoritmo en función de la cantidad de elementos que tienen un peso mayor a W (para $W = 50$ y $W = 200$. Se puede comprobar viendo el gráfico que, para los dos pesos que se tuvieron en cuenta, hay un decremento de las llamadas recursivas a medida que aumenta la cantidad de ítems que no son factibles de considerar para armar una solución, o sea que efectivamente se podan soluciones.

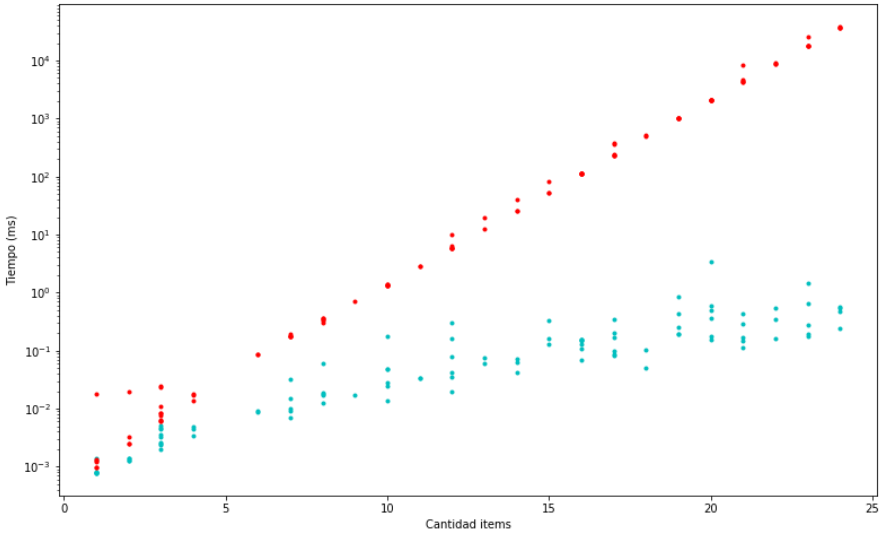
Tiempo en función de la cantidad de ítems descartables ($n = 19$, $W = 50, 200$)



Cantidad de llamadas recursivas en función de la cantidad de ítems descartables ($n = 19$, $W = 50, 200$)



Otro test que se hizo fue generar 100 instancias del problema de la mochila para $W = 100$ (por cuestiones de tiempo no se llegó a probar para distintos W) y n no mayor 25 y correr dicho test con fuerza bruta y backtracking. Se agrega el siguiente gráfico en el que se muestran los resultados de ambos algoritmos para el test utilizando una escala logarítmica para poder comparar mejor los órdenes de complejidad. El algoritmo de backtracking con poda es irregular debido a que los datos de entrada testeados son pseudoaleatorios y pueden tener mejor o peor comportamiento frente a la poda por factibilidad. Es notoria la mejora en términos de tiempos de cómputo del backtracking con poda en contraste con el backtracking sin podar utilizado en fuerza bruta.

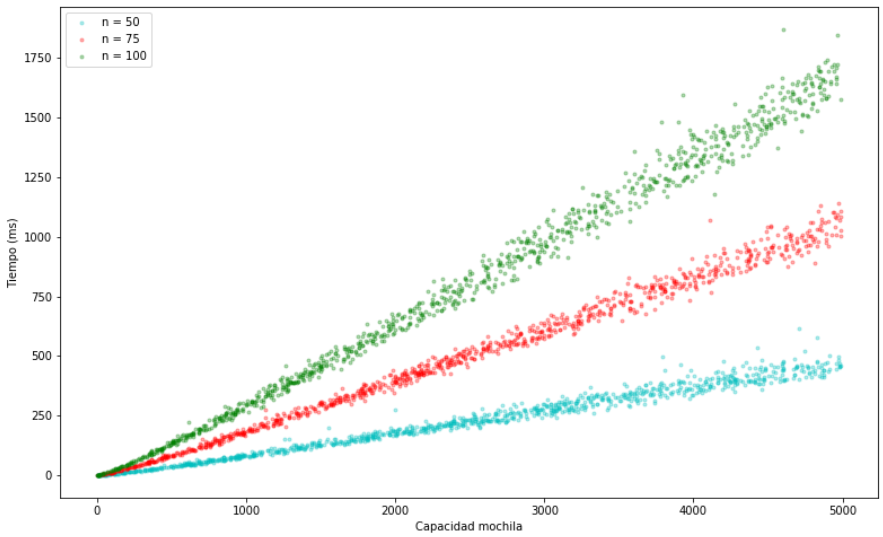


7. Programación Dinámica

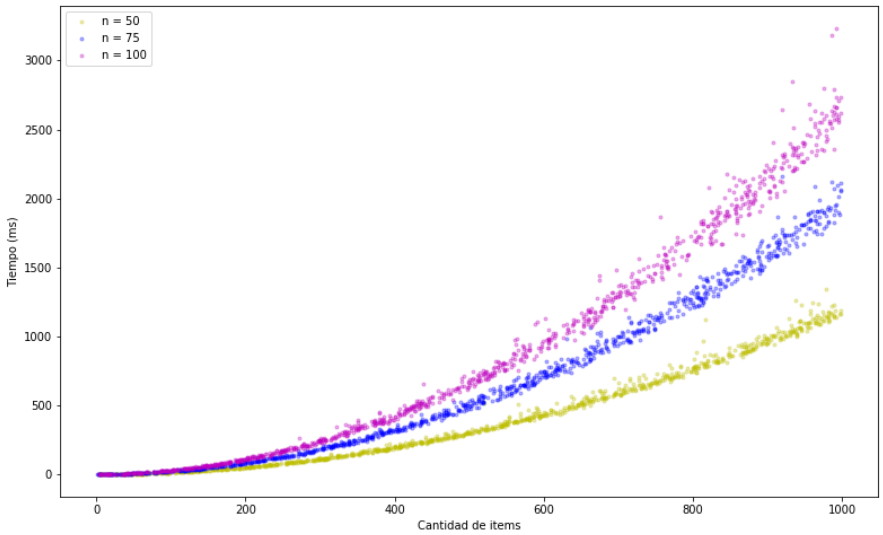
Para este algoritmo se consideraron dos tests generando 1000 instancias para cada uno. Para el primero se deja n fijo (probando para $n = 50, 75, 100$) y se hace variar el W con un valor no superior a 5000. En el segundo se deja W fijo (para los valores $W = 50, 75, 100$) y se hace variar n con un valor no superior a 1000. Dado que la complejidad temporal del algoritmo es $\mathcal{O}(nW)$, se espera que los gráficos obtenidos los tests en donde se deja fija una de las variables n, W y se varía la otra, se asemejen a una recta. En el experimento se midió el tiempo que tarda el algoritmo para procesar cada una de estas instancias.

Ambos gráficos expresan el tiempo en milisegundos en relación a la capacidad de la mochila o cantidad de items según corresponda. Respecto al primer gráfico, se observa que si bien para cada uno de los n, el gráfico se asimila a una recta, cuanto más se aumenta el W más varianza de datos hay, esto se cree que es porque si bien la complejidad temporal del algoritmo es nW, también entra en juego la memoria, si se usa mucho este recurso llega un punto en que no hay espacio en la RAM para seguir operando y se usa memoria del disco causando un gran retardo. Con respecto al segundo gráfico, no se puede comprobar lo que habíamos asumido, si bien para el caso $W = 50$, a partir de $n = 600$ se puede ver una tendencia de los puntos a acumularse alrededor de una recta, para $W = 75$ y $W = 100$ globalmente los gráficos se asemejan más a algo de tipo exponencial. No alcanzó el tiempo para arrojar más datos y poder llegar a algo más concluyente.

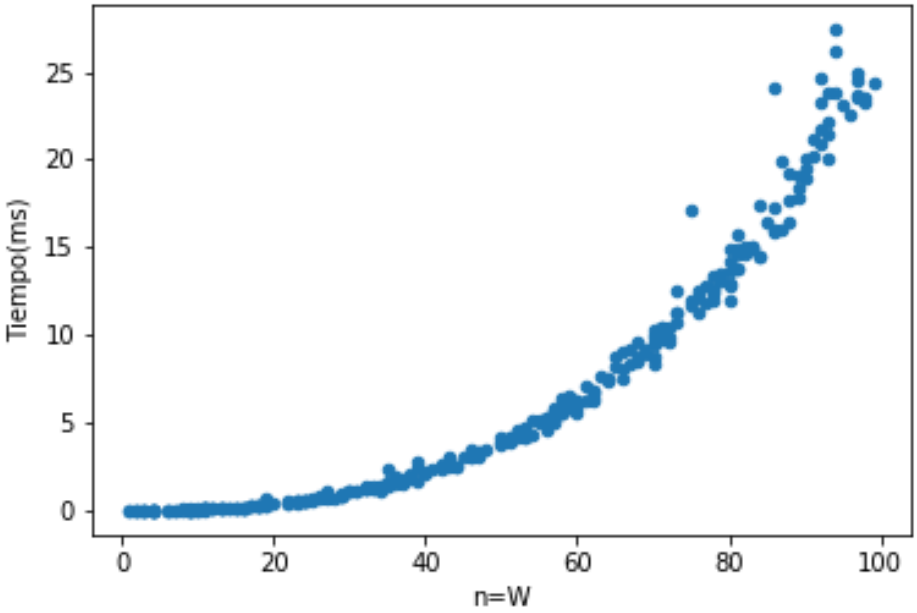
Programación Dinámica para W variable, n = 50,75,100 fijo



Programación Dinámica para W=50,75,100 fijo, n variable



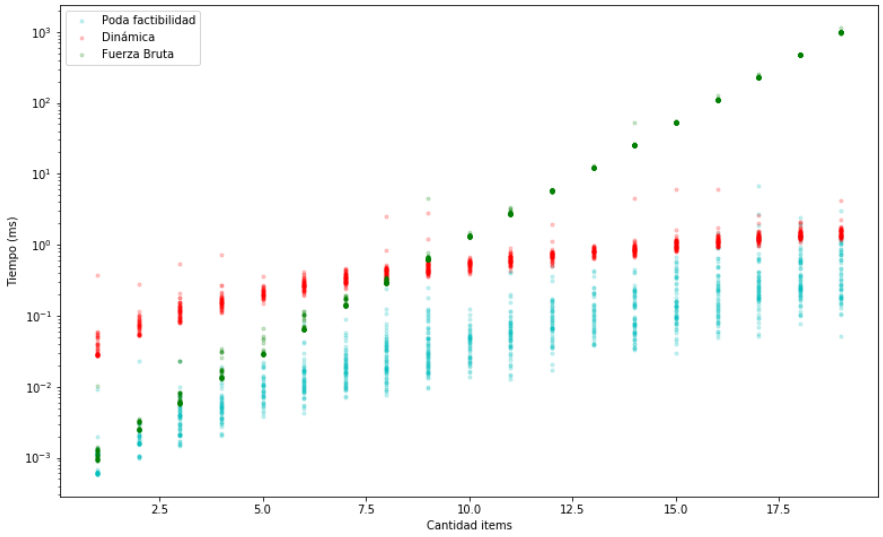
El último test que se hizo relacionado al algoritmo de programación dinámica fue generar 300 instancias de manera aleatoria pero con la condición de $n = W$ para valores no superiores a 100. Se presume que, dado que la complejidad teórica temporal de programación dinámica es $\mathcal{O}(nW)$, si ambos valores son iguales, entonces $\mathcal{O}(nW) = \mathcal{O}(n^2)$, por lo que consideramos ver empíricamente si esto se comprobaba. A continuación se expone el gráfico del tiempo de cómputo de algoritmo (en milisegundos) en relación a la cantidad de items/capacidad de mochila (ambos son los mismos). Efectivamente, se puede ver la semejanza de la función con un polinomio cuadrático, por lo que se comprueba la hipótesis construida.



Parte IV
Discusión y conclusiones

8. Análisis comparativo entre algoritmos

A modo de conclusión se presenta un gráfico con el tiempo de ejecución de los tres algoritmos para 1000 casos generados pseudeoaleatoriamente con instancias no mayores a 20 por limitaciones de tiempo y para $W = 100$ y una breve discusión de ellos según lo analizado en teoría y observado en la experimentación.



El estudio comparativo de estos tres algoritmos nos muestra que, si bien sus complejidades teóricas son conocidas, la naturaleza del problema al cual estos están siendo aplicados determina que algunos sean más útiles que otros más allá de que en teoría una complejidad parezca a priori peor que otra. En el caso del problema de la mochila, backtracking con poda de factibilidad parece ser el que mejor tiempos de ejecución tiene de los tres. Por otro lado, hasta $n = 10$ dinámica es el algoritmo con mayor tiempo que los otros dos pero a partir de $n = 10$, ya fuerza bruta que tanto mejor como peor caso es exponencial, lo sobrepasa. Esto tiene sentido puesto que justamente si $W = 100$, se tiene que $100n < 2^n$ a partir de tal valor. Por otra parte, vemos también que el que más dispersión de datos tiene para un n fijo es backtracking con poda, lo cual es lógico puesto que este algoritmo es el que depende fuertemente no solamente de la cantidad de elementos si no que también, al momento de podar soluciones, juega un fuerte rol la relación entre el W y los pesos de los ítems; puesto que en este test se consideraron instancias con mismo n pero distinto peso y beneficio , backtracking varía su tiempo de ejecución dependiendo de esto.