TP1 ALGO3

Introducción
Alumnos y libreta:
Julián Recalde camos (502/17)
Juan Fernandez Zaragoza (839/17)

El problema de la Mochila

El problema de la Mochila, o Knapsack Problem, es un problema de optimización sobre un conjunto de elementos. Dado un conjunto de objetos que pueden elegirse, se debe elegir el mejor subconjunto según las siguientes condiciones:

-Cada objeto lleva asociados dos valores. Uno representa su costo y otro su beneficio. En el problema de la mochila, se llama "peso" al costo y "valor" al beneficio.

-Se cuenta con una cota superior al costo aceptable. No se permite que la suma de los costos de los objetos del subconjunto a elegir supere esa cota. La mochila tiene un umbral de peso máximo que puede sostener.

-El subconjunto de elementos escogido debe maximizar la sumatoria del beneficio de los objetos seleccionados, sin que la sumatoria de sus costos supere el umbral dado.

El problema que trabajamos aquí es un problema muy similar, en que no es necesario devolver el subconjunto explícitamente, sino solamente el valor máximo de beneficio total que una elección óptima de ítems pudiera alcanzar, con las mismas condiciones. Dada una cota máxima de costo y un conjunto de pares ordenados donde el primer elemento del par representa el costo y el segundo el beneficio, se debe devolver el valor entero del beneficio del mejor conjunto.

Por ejemplo, con
cota = 10
lista = [(2, 4), (3, 9), (5, 5), (7,3)]
debe devolverse 18, que es el beneficio total del subconjunto óptimo [(2,4),(3,9),(5,5)]

y con
cota = 20
lista = [(9,17), (12, 33), (11, 2)]
debe devolverse 33, que es el beneficio total del subconjunto óptimo [(12, 33)]

Los datos no se presentan en forma de listas sino como un archivo de entrada en que cada línea sucesiva contiene los valores que representan al elemento, separados por un espacio. La primera línea contiene la cota y un entero que representa la cantidad de ítems disponibles.

Fuerza Bruta

La forma más directa de solucionar el problema es por fuerza bruta, pero también es la más lenta. Intuitivamente, lo que haría un algoritmo de fuerza bruta en este caso es calcular el resultado para cada uno de los subconjuntos posibles, y luego escoger, de todos ellos, el mÃ¡ximo beneficio de entre aquellos que no superan la cota de peso. Para ello, entonces, debe contarse con, para cada uno de los subconjuntos posibles, la tupla que corresponde a su peso total y a su valor total.

El algoritmo de solución se divide entonces en dos etapas: una primera, correspondiente a la generación de la lista que representa los resultados del conjunto potencia, y una segunda, que recorre dicha lista y devuelve el valor óptimo.

La primera etapa puede resolverse recursivamente.

-Caso base: Si la lista está vacía (es decir, si no hay elementos para escoger) se devuelve una lista que solamente contiene a la tupla (0,0).

-Caso recursivo: Si la lista no está vacía, se selecciona un ítem c. Se resuelve luego la lista para el conjunto de elementos sin c (lista - {c}). Luego, al resultado de esta lista, se le debe agregar todas las tuplas correspondientes a c+i, representando c+i todas las tuplas que sean de la forma (peso(c) + peso(i), valor(c) + valor(i)), para todo i pertenenciente a la lista.

La solución es similar a la de formar un conjunto potencia, pero en lugar de elegir un ítem e introducirlo en los subconjuntos para luego unir el conjunto resultante con el conjunto de los subconjuntos sin el ítem, se suma sólo los valores de c al resultado de cada uno, lo cual representa el mismo principio. Dado que se sigue el mismo principio de recursión que la recursión correspondiente al conjunto potencia, adaptada al problema particular, no es difícil ver por qué funciona: a efectos de este problema, el agregado del peso y valor del ítem separado al peso y valor de algún subconjunto hace las veces del agregado del ítem separado al subconjunto en los casos de evaluación del conjunto potencia.

Se implementan entonces dos funciones para llevar a cabo esta etapa. Siendo C un conjunto de ítems, A y B son conjuntos de tuplas que representan subconjuntos y c un ítem (a efectos de implementación y del algoritmo, no hay diferencia entre los elementos de A y de C, la diferenciación es útil para aclarar qué se está haciendo).

```
sumaspartes(C):
        si el conjunto es vacío,
                devuelve la lista con el (0,0)
        si el conjunto no lo es
                separa un elemento c
                calcula sumaspartes(C-c)
                devuelve valorespartes(sumaspartes(C-c), c)
```

Donde valorespartes implementa el caso recursivo

```
valorespartes(B, c):
        genera un conjunto A vacío
        mientras B no está vacío hace:
                quita un elemento b de B
                agrega b a A
                agrega b+c a A
        devuelve A
```

La segunda etapa se implementa con la funciÃ³n elijooptima donde A es un conjunto de tuplas que representan subconjuntos y n es un entero que representa la cota de peso.

```
elijooptima(A, n)
        genera una tupla variable max, inicializada en (0,0)
        mientras A no está vacío hace:
                quita un elemento b de A
                si peso(b) <= n y valor(b) > max entonces max = b
        devuelve max
```

Para resolver el problema para un conjunto de objetos C y una cota n debe correrse:

```
elijooptima(sumaspartes(C), n)
```

La complejidad del algoritmo es de 2^n, porque hay n niveles de llamados recursivos, y en cada nivel se duplica el tamaño de la lista de partes formada. El llamado recursivo más pesado es el último en computar, y es cuando la lista recibida tiene tamaño 2^n-1 y la lista resultante tiene tamaño 2^n que es el resultado de añadir a todos los subconjuntos computados con los correspondientes agregados del primer elemento. Por esta razón, el orden de complejidad del peor caso es de n*2^n, donde n es la cantidad de elementos del input. La recorrida final para encontrar el óptimo es lineal respecto del tamaño del conjunto de partes, que es 2^n, por lo que no añade complejidad en términos asintóticos.

Meet in the middle

La primera estrategia implementada que reduce el tiempo de ejecución del algoritmo de fuerza bruta es el de Meet in the Middle. La idea de dicho algoritmo es separar el problema en dos subproblemas más simples y a partir de los resultados de ambos dar la solución al problema general.

En este caso, se divide la lista de entrada en dos listas de igual tamaño. A continuación, se devuelve la lista de resultados para cada una (la lista de valores y pesos correspondiente a cada subconjunto de los resultantes de la división).

En tercer lugar, debe darse la solución al problema general a partir de estas dos listas.

Sabemos que cualquier subconjunto de la primer mitad tiene una intersección vacía con cualquier subconjunto de la segunda mitad. Además, sabemos que cualquier subconjunto del conjunto inicial puede ser formado por la unión de algún subconjunto de la primera mitad con algún subconjunto de la segunda mitad. Por lo tanto, el subconjunto óptimo del conjunto total será la mejor combinación entre un subconjunto del primero y uno del segundo.

Para optimizar la búsqueda de la mejor combinación, se ordena por peso la primera mitad. Luego, se eliminan de dicha lista todos los subconjuntos que tengan el mismo peso que otro pero menor valor, o mayor peso pero menor valor (sabemos que no formarán parte de la solución, porque si peso(A) >= peso(B), B puede combinar con cualquier subconjunto C de la otra mitad con que pueda combinar A, significando esto que si C es parte de la solución, A no puede serlo, porque valor(C+B)>=valor(C+A).

Tras hacer esto, se busca para cada elemento de la segunda mitad una pareja de la primera lista. Dado que en la primera lista los elementos de mayor peso tendrán mayor valor (se eliminaron los inútiles), basta con encontrar el elemento de mayor peso que sumado al peso del subconjunto a combinar mantengan un peso menor al de la cota. Dado que la lista está ordenada por peso, esta búsqueda puede ser binaria.

Una vez que se encontraron las parejas para todos los elementos del segundo conjunto, basta solamente encontrar la mejor pareja, y eso devuelve el óptimo.

Entonces, el procedimiento sería:

```
dividirmitades(C):
        incializa entero n = tamaño(C)/2
        declara lista A
        declara lista B
        mientras C no esté vacío hace:
                si tamaño(C) > n, añade primero(C) a A
                si no, añade primero(C) a B
                quita el primer elemento de C
        devuelve [A, B]
```

Luego se ordenan A y B usando mergesort, de complejidad O(n log n), respecto del peso (comparando el primer elemento de las tuplas).

A continuación, se quitan los elementos "inútiles" de ambas listas ordenadas con la función sacoinutiles, que toma una lista ordenada y devuelve un vector ordenado sin inútiles (transforma de lista a vector para luego poder hacer búsqueda binaria).

```
sacoinutiles(A)
        declara vector vacío res
        si tamaño(A) == 1, devuelve res<-primero(A)
        si no, mientras tamaño(A) > 1
                iterador it al inicio de A
                it++
                si valor(primero(A)) >= valor(*it)
                        eliminar(it)
                        si tamaño(A) == 1 añade el primero de A a res // no queda nada por comparar
                si no,
                        si el primero pesa igual que el segundo, elimina el primero de A
                        si no, añade el primero de A a res
                        elimina el primero de A
                        si tamaño(A) == 1, añade el nuevo primero también a res

        devuelve res
```

Luego se define la función que, dada una tupla (que representa un subconjunto en una de las listas) encuentra su pareja adecuada en el vector que representa a la otra, teniendo en cuenta la cota de peso. Notar que basta agarrar la más pesada que sea menor a cotapeso - peso(A), porque el vector está ordenado por peso y sin inútiles. Notar también que el vector B no puede ser vacío ya que siempre tiene el elemento <0, 0>

```
buscarpareja(a, B, cotapeso)
        declara entero n = tamaño(B)
        declara entero bottom = 0
        declara entero top = tamaño(B)-1
        declara pesopermitido = cotapeso - peso(a)
```

```
si pesopermitido <= 0, pareja = B[0]
si no,
        mientras top - bottom > 1 hacer:
                declara entero medio = (top +bottom)/2
                si peso(B[medio]) > pesopermitido
                        top = medio
                si no, bottom = medio
        si peso(B[top]) <= pesopermitido devuelve B[top]
        si no, devuelve B[bottom]
```

luego arma la lista de mejores parejas para cada elemento de A, recorriendo A, buscando su pareja en B, y agregando para cada elemento de A, la tupla resultante de sumarla a su pareja en la lista de salida.

Finalmente recorre dicha lista de salida buscando el máximo del peso, haciendo una búsqueda lineal.

La complejidad de meet in the middle es siempre de n*2^n/2. Esto es porque la función que arma mitades toma tiempo lineal sobre n (lee linealmente y crea las dos mitades), luego calcula para cada mitad el conjunto de partes acorde a fuerza bruta (que tomará 2^n/2 cada uno, porque cada una de ellas tiene la mitad de tamaño que n, y el algoritmo de fuerza bruta es exponencial sobre el tamaño del input en base dos). Luego, ordenar una de las listas, con mergesort toma m*log(m), donde m es el tamaño de la lista a ordenar, que es 2^n/2. Pero 2^n/2*log(2^n/2) es igual a 2^n/2*n/2, porque el logaritmo se cancela. Sacar inútiles es lineal respecto de 2^n/2 porque recorre linealmente la lista, y la búsqueda de pareja para cada uno de los elementos de una lista toma log(2^n/2) porque es búsqueda binaria, y como se hace linealmente para cada uno de los elementos de la lista, la complejidad resultante es de n*log(2^n/2). El conjunto de parejas es de tamaño 2^n/2, y encontrar el máximo toma tiempo lineal respecto de eso. Dado que todas estas complejidades están acotada superiormente por n*2^n/2 y se ejecutan secuencialmente una finita cantidad de veces, el tiempo del algoritmo está acotado por alguna constante k por n*2^n/2, lo cual resulta en una complejidad asintótica de O(2*2^n/2).

Programación dinámica

El enfoque en programación dinámica tiene un procedimiento similar al recursivo, pero teniendo en cuenta lo siguiente: Dado que muchas veces los subproblemas resueltos son iguales, el algoritmo puede ahorrarse tiempo de ejecución si guarda los resultados de los subproblemas, evitando así la ejecución redundante de instrucciones.

Para esto, se implementa una solución bottom-up, utilizando una matriz en que las filas representen los ítems y las columnas el peso máximo. La matriz se llena de ceros y luego se corre un algoritmo que calcula los valores para cada celda teniendo en cuenta lo siguiente:

Dado un elemento del arreglo, este puede estar o no en el subconjunto óptimo. Si lo está, entonces lo hace junto a la mejor mochila que pueda armarse en la lista (exceptuando al propio objeto) y con la cota de peso siendo la cota anterior menos el peso del objeto. Si no lo está, entonces la solución óptima será la mejor mochila exceptuando al objeto. Habiendo resuelto ambos subproblemas, se debe calcular cuál es la mejor mochila resultante, y esa será la mejor de las posibles.

Se forma la matriz teniendo en cuenta eso, y teniendo en cuenta que la celda iw de la matriz contiene el valor de la mejor mochila con cota de peso w que toma en cuenta hasta el ítem i (es decir, hace lo mismo que el algoritmo recursivo, pero bottom up).

Al algoritmo se le pasa el vector de elementos C y la cota de peso n, y este hace lo siguiente:

```
matrizknapsack(C, n)
        declara una matriz de tamaño n+1*tamaño(C)+1
        llena de ceros las celdas correspondientes al peso cero
        desde i == 1 hasta i == tamaño(C)
                y desde w == 1 hasta w == n hace
                        si i == 0 o w == 0, entonces M[w][i] = 0
                        si no, si peso(C[i-1])<= w entonces el ítem es candidato, y por lo tanto hace
                                declara entero sinprimero = M[w][i-1]
                                declara entero conprimero = valor(C[i-1]) + M[w - peso(C[i-1])]
                                M[w][i] = max(sinprimero, conprimero)

                                si no, el item no es candidato y por lo tanto    M[w][i] = M[w][i-1]
```

devuelve M[n][tamaño(C)], que es la mejor mochila con cota n y todos los ítems disponibles (siguiendo el invariante de las iteraciones).

La complejidad de la programación dinámica debería ser de O(w*n), donde w es la cota de peso y n el tamaño del input, en todos los casos, porque siempre se construye la matriz entera. Arma una fila para cada entero entre 0 y w. Dado que los accesos a la matriz y al arreglo pueden hacerse en O(1), cada ciclo del while toma tiempo constante, y se hacen w*n ciclos. Por esta razón, con w constante, la complejidad es lineal respecto del tamaño del input. Sin embargo, el tiempo de ejecución del algoritmo varía respecto de w, con lo cual, para n pequeños y w grandes el algoritmo puede no ser el mejor.

Backtracking

El objetivo del algoritmo de backtracking es dar una lista similar a la de fuerza bruta de subconjuntos posibles, para después elegir el óptimo entre ellos. Sin embargo, tiene en cuenta el siguiente punto: Dado que cada subconjunto se obtiene tomando otro subconjunto más pequeño y agregándole cero o más elementos, hay algunos subconjuntos que no deben formarse. Es sabido que hay conjuntos de elementos que no pueden ser parte de la solución. Una razón posible para esto es que en sí ya superan la cota de peso: en ese caso, ningún agregado de más objetos va a volverlo un subconjunto susceptible de ser el mejor, porque la cota de peso ya ha sido superada. Por lo tanto, no debe tenérselo en cuenta en la producción de más subconjuntos (se poda toda una rama de formación de subconjuntos posibles). Otra razón puede ser que el subconjunto del que se está hablando no pueda ser el óptimo porque no está en condiciones de superar a otro subconjunto que ya se analizó. Si se guarda el valor máximo de los subconjuntos analizados hasta el momento, basta comparar el subconjunto a formar con el máximo, de la siguiente manera: si la ganancia disponible para el conjunto a analizar más la ganancia del propio conjunto es menor que la ganancia de un máximo anterior, el conjunto actual no puede ser un subconjunto del óptimo, porque con nada que se le agregue va a poder llegar a superar al otro candidato.

La ganancia disponible para un conjunto se calcula de la siguiente manera:

Se calcula primero la ganancia total potencialemnte posible sumando los valores de todos los ítems de la lista. Cada vez que se analiza un ítem que pudiera agregarse al conjunto, su valor se resta de la ganancia disponible (tanto si se agrega al subconjunto como si no, el resto de los ítems que pueden ser agregados reducen el total de ganancia posible cuando se quita un elemento de los elementos posibles a agregar).

El algoritmo es muy similar, luego, al de fuerza bruta, pero con dos modificaciones. La función valorespartes, que toma un conjunto y un elemento para construir una solución parcial tiene una nueva guarda correspondiente a la poda. Se añade una guarda al algoritmo que contempla lo dicho anteriormente:

Cuando se recorre el conjunto que corresponde a las partes, si en fuerza bruta se añadían, para cada subconjunto, el mismo subconjunto y el subconjunto agregando el nuevo elemento, ahora hay una guarda que decide no agregarlo, así podando todas las posibles añadidas futuras (el algoritmo es recursivo). Para esto, se compara el peso del subconjunto resultante con la cota de peso, y se compara el valor del nuevo subconjunto añadido al valor restante (aun no computado) para dicho subconjunto. Si el valor potencial es menor al anterior máximo o si la cota de peso es menor al peso del subconjunto resultante, el subconjunto no se añade, efectuando así la poda.

Para esto, en cada paso se debe guardar el subconjunto máximo parcial, haciendo una comparación entre el anterior máximo y el valor del conjunto que se está considerando como respuesta.

En el llamado recursivo, entonces, se tienen en cuenta también la cota de peso, el ganador actual y la sumatoria de valor aún no computable (valor disponible).

La cota superior de complejidad puede ser la misma que la de fuerza bruta, porque todas las comparaciones que se agregan pueden hacerse en O(1), puesto que, como se ha dicho, se comparan variables almacenadas y accesibles. El cálculo del primer valor de ganancia total toma tiempo lineal respecto del tamaño de la lista del input, y la actualización de esos cambios se hace con la resta mencionada, en O(1). Ahora bien, la complejidad promedio debería reducirse, porque las podas pueden reducir la cantidad de elementos en cada paso sucesivo. Si hay una poda, en ese nivel no se llega a duplicar la cantidad de elementos que había en el nivel anterior. En promedio, asumimos que habrá podas, por lo que el tiempo de ejecución promedio debería ser inferior al promedio de fuerza bruta, que siempre analiza todos los casos.

Experimentación:

Observaciones iniciales:

Teniendo en cuenta las complejidades teóricas de cada uno de los algoritmos, observamos lo siguiente: 1) salvo programación dinámica, todas las complejidades teóricas del peor caso de los algoritmos implementados dependen solamente de n. 2) Fuerza bruta y meet in the middle son dos algoritmos cuya complejidad no varía entre el mejor y el peor caso. 3) Por otra parte, backtracking podría tener casos de menor

complejidad, dado que puede contener podas. 4) Por último, vemos que manteniendo una cota de peso constante, la complejidad del algoritmo de programación dinámica crece linealmente respecto de n y que 5) Programación dinámica se ejecuta en tiempo pseudopolinómico, si se tiene en cuenta el tamaño en bits de w.
Pregunta:

Dado que existen diferencias en las complejidades de distintos algoritmos, nos preguntamos lo siguiente: ¿Tendrán efecto estas diferencias en la implementación real, generando diferencias significativas en el tiempo de ejecución? Y de ser así, ¿qué algoritmo funcionará mejor para cada tipo de input, si variáramos la cota de peso, el tamaño de la colección y el orden de los elementos?

Hipótesis:

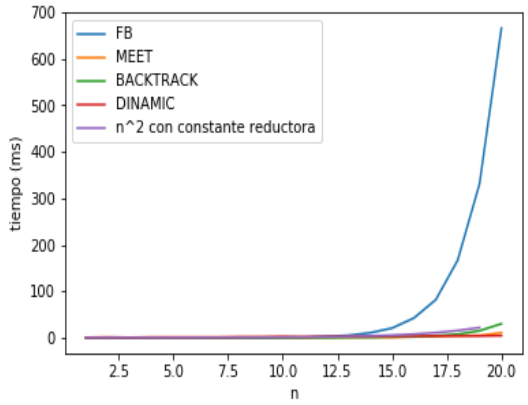
Consideramos que fuerza bruta tomará siempre su cota máxima temporal de O(n*2^n), y meet in the middle tomará, también, siempre lo mismo que en su peor caso, es decir, O(n*2^n/2)). Con lo cual, consideramos que meet in the middle siempre variará mejor respecto de n que fuerza bruta, y que para ninguno el peso variará si se cambia el orden o los valores de los elementos.

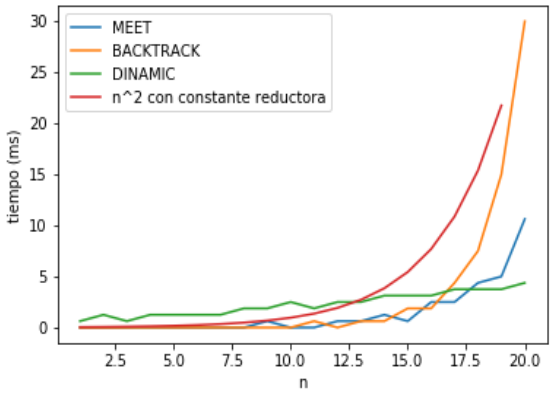
Dado que programación dinámica es el único que varía su complejidad respecto de w, y que manteniendo un w constante es el que mejor varía respecto de n, consideramos que, para w constante y n creciente será el algoritmo que mejor variará asintóticamente respecto de n, pero que para n pequeños y w grandes será el peor algoritmo. Dado que se recorre toda la matriz, y para cada celda se resuelve en O(1), no consideramos que el orden de los elementos ni sus valores modificarán significativamente el tiempo de ejecución.

Backtracking es el algoritmo que más puede variar, en términos de tiempo de ejecución respecto de los valores y el orden de los elementos en el input, por las podas. Aunque en su peor caso tomaría la misma cantidad de tiempo que fuerza bruta, las podas por cota de peso y por optimalidad pueden mejorar su rendimiento. En su peor caso, debería tomar más que meet in the middle. En general, debería tomar menos. Mejorará su rendimiento con cotas de peso reducidas (porque esto incrementa el número de podas por peso) y también mejorará si los elementos están ordenados de menor valor a mayor valor (porque esto aumenta la cantidad de podas por optimalidad).

Experimentos:

Caso 1: Ordenado de mayor a menor, con n variando hasta 20 y w constante en 3500. Los números de máximo 5 cifras y cota de peso de cuatro cifras.
en OrdenadomayoramenorN20W3500 se encuentra las tablas originales

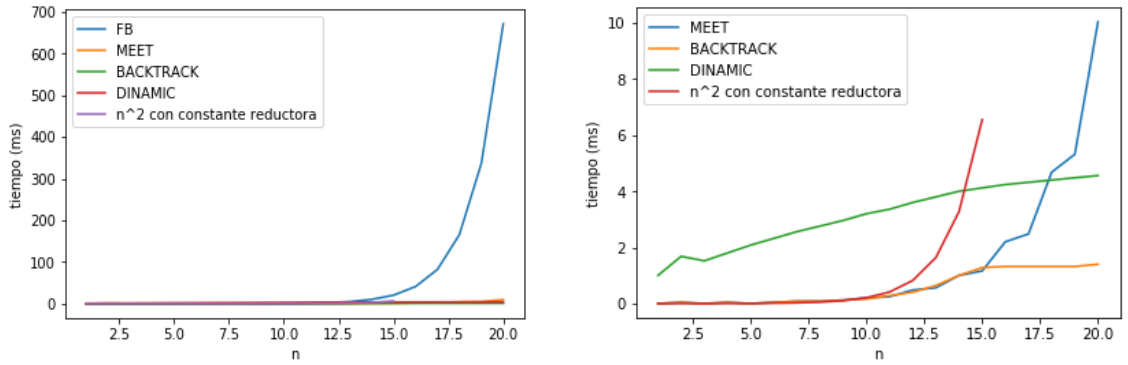




En este experimento se corroboran las siguientes hipótesis:

-Meet in the middle es mejor, en términos temporales, que fuerza bruta
-Dinámica, con w constante, varía mejor respecto de n (pero para n más chicos es peor)
-Backtracking fue peor que meet in the middle hasta n=20 (suponemos que es porque, como hipotetizamos, este es el peor ordenamiento para backtracking en términos de optimalidad). Sin embargo, sigue siendo mejor que fuerza bruta porque la cota de peso elimina varios elementos.

Caso 2: Ordenado de menor a mayor, con n variando hasta 20 y w constante 3500.
los datos se encuentran en OrdenadomenormayorN20W3500



En este experimento los valores de los elementos son iguales que en el anterior, así como la cantidad de elementos en la colección y la cota de peso, pero los elementos están en el orden inverso. Corrobora ciertas hipótesis, en conjunción con el caso 1, respecto de cómo afecta el orden de los elementos a los distintos algoritmos.

Tabla comparativa
Una tabla que compara el valor final (n=20) en cada uno de los gráficos puede ser útil para visualizarlo mejor:

	Tiempo final caso 1 (ms)	tiempo final caso 2 (ms)
Backtracking	29.952	1.401
Dinámica	4.368	4.560
Meet in the middle	10.608	10.041
Fuerza bruta	665.713	671.318

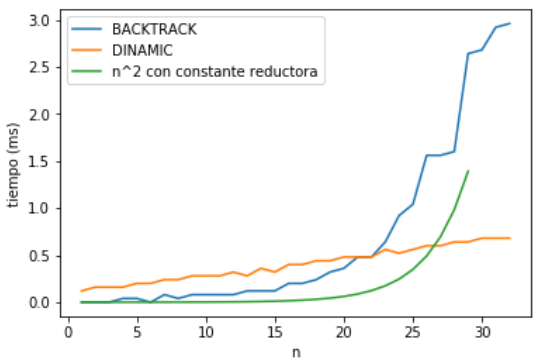
-Ni meet in the middle ni fuerza bruta ni dinámica modificaron su rendimiento apreciablemente (sus curvas en ambos gráficos terminaron en valores similares).

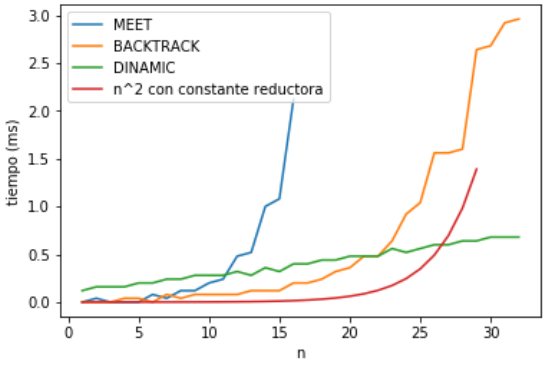
-Backtracking mejoró significativamente su rendimiento, superando a los demás en economía temporal. Como supusimos en la hipótesis, este orden es mejor para dicho algoritmo, que es el único en que podemos ver, respecto del orden, una significativa variación en sus tiempos.

-Puede apreciarse, además, que el algoritmo de programación dinámica, manteniendo w constante, funciona más lentamente para valores de n pequeños (superando a backtrack y a meet in the middle).

Caso 3: Valores aleatorios, n variando hasta 32, con w constante 500, números de 3 dígitos.

los datos se encuentran en Random3DIGITOSN32W500



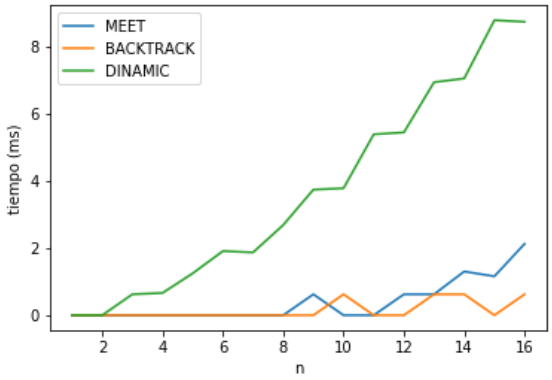


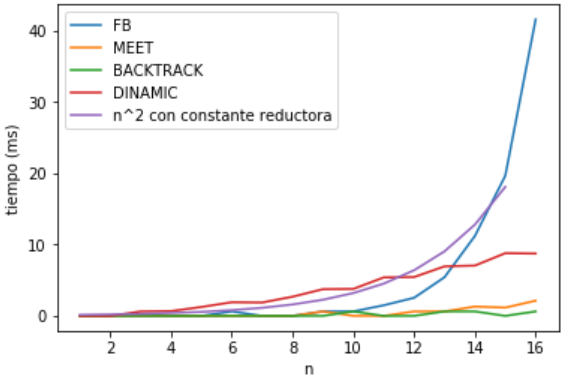
-Vuelve a verse el comportamiento de programación dinámica respecto de los otros algoritmos. Con w constante, para n chicos, corre peor, pero para n más grandes tiene una curva favorable. (Es lineal contra otros cuyas curvas tienden a ser exponenciales).

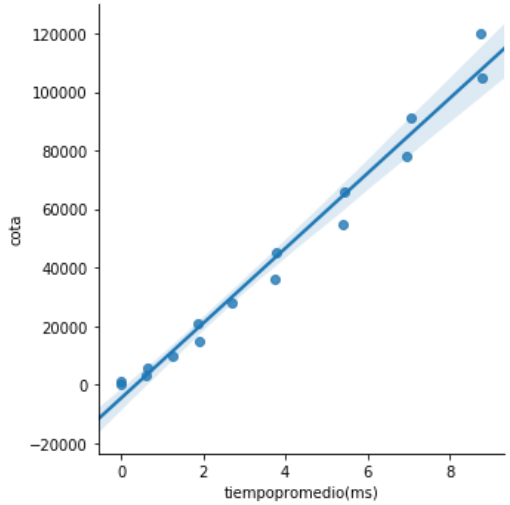
-Al hacer el experimento, fuerza bruta tomaba demasiado tiempo en la compilación. No está graficado, pero podemos ver que para este caso backtracking sigue siendo mejor que fuerza bruta (en un caso promedio, random, se aleja de su peor caso que es el ordenamiento de mayor a menor). Vemos también, como lo supusimos, que meet in the middle también tomó menos que fuerza bruta.

-Para n chicos, meet in the middle tardó menos que backtracking, mientras que para n grandes sucedió lo contrario. Esto es porque, en general, se incrementa el número de podas cuando aumenta la cantidad de elementos si se mantiene w constante. Dado que este es un caso promedio para backtracking (se aleja de su peor caso en el que se asemeja a fuerza bruta), y meet in the middle no varía respecto del orden, esta diferencia se va volviendo más significativa cuando n crece.

Caso 4: Varían n y w, n de 0 a 16, w aumentando de a 500, de 0 a 7500. Muestra cómo dinámica es peor que los otros algoritmos cuando aumenta w.
los archivos se pueden encontrar en ordenadoNi=0 f=16 Wi=0+500 2 variables





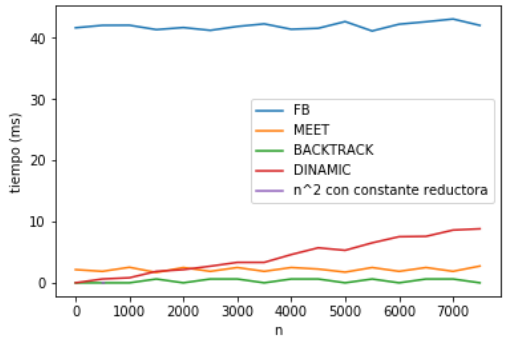


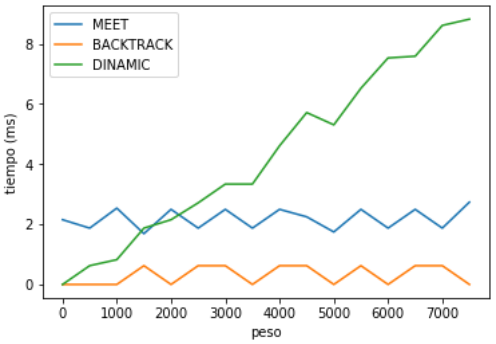
Correlación dinámica y n*w:

A la izquierda gráfico correlación, con r=0.9898183392665, que muestra que la complejidad real de programación dinámica se correlaciona bien con la esperada.

En los gráficos del caso 4 (arriba), vemos cómo con n y w variables crecientes, dinámica funciona peor que backtracking y meet in the middle. Sin embargo, en este caso, fuerza bruta sigue funcionando peor en términos personales.

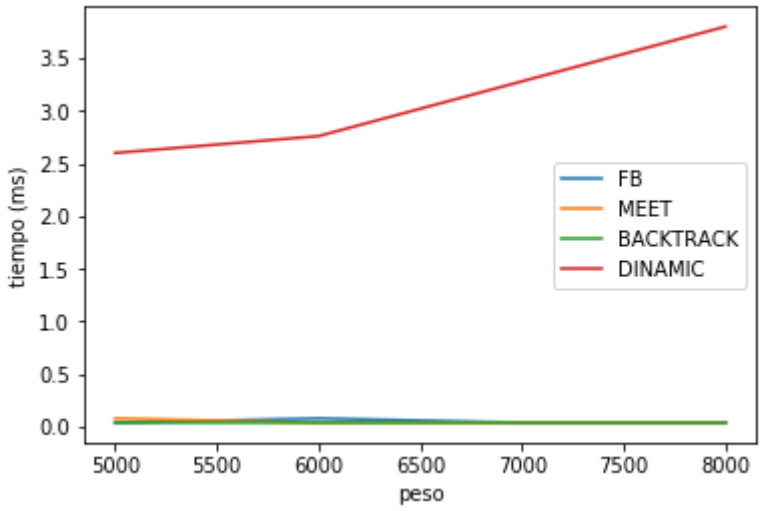
Caso 5:
se encuentra en OrdenadoN16 Wi=0+500 variable
n=16, w empieza en cero y aumenta de a 500 hasta 7500
n constante, w creciente





-dinámica varía mucho de la manera hipotetizada
-se ve cómo ninguno de los otros tres varía significativamente.
-dinámica, en este caso, sigue siendo mejor que fuerza bruta.

Caso 6: N constante = 4, w =5000 + 1000 hasta 8000 (aca el peso es variable creciente y esta fijo n en 4 elementos, y w empieza en 5000 y aumenta de a 1000 llegando hasta 8000)
datos se pueden encontrar en InputN=4W=5000 +1000 WF=8000



En este caso podemos ver que programación dinámica puede ser peor que fuerza bruta (porque es el único cuyo tiempo varía significativamente respecto de w.

Conclusión:

Los experimentos realizados han corroborado nuestras hipótesis. En casos de w muy grande, manteniendo n constante, programación dinámica puede ser más costoso que fuerza bruta. Sin embargo, en general, fuerza bruta es el algoritmo más costoso. Ni meet in the middle ni fuerza bruta ni programación dinámica varían significativamente su costo temporal al alterar el orden de los elementos en el input. Sin embargo, el costo de backtracking sí es susceptible a estas modificaciones, de la manera en que lo hemos predicho. También pudimos ver que backtracking funciona mejor con valores de w relativamente bajos, y que con w constante, y en casos promedio, para valores grandes de n funciona mejor que meet in the middle. Por último, hemos podido ver que para valores de w constantes, el que mejor varía respecto de n es programación dinámica. Hemos visto cómo las variables estudiadas afectan el tiempo de ejecución de los algoritmos, mejorando así nuestra capacidad de decidir, para distintos casos, qué algoritmo es conveniente usar.

Apéndice:
A modo de apéndice, mostramos las tablas de valores de los dos primeros experimentos. El resto puede encontrarse en el archivo, por cuestiones de espacio no han podido ser incluidas aquí. Además, en la carpeta casos test se encuentran más experimentos que no hemos podido añadir por falta de espacio.

Caso 1 tabla:

Fuerzabruta

n	peso	tiempopromedio(ms)	ganancia maxima
0	1 3500	0.00000	0
1	2 3500	0.00000	0
2	3 3500	0.00000	0
3	4 3500	0.00000	0
4	5 3500	0.00000	3100
5	6 3500	0.00000	3100
6	7 3500	0.00000	3100
7	8 3500	0.62404	3100
8	9 3500	0.00000	3500
9	10 3500	0.62400	3500
10	11 3500	0.62400	3500
11	12 3500	2.49600	3500
12	13 3500	4.99200	3500
13	14 3500	10.60800	3500
14	15 3500	20.59200	3500
15	16 3500	42.43210	3500
16	17 3500	81.74420	3500
17	18 3500	166.60800	3500
18	19 3500	331.08200	3500
19	20 3500	665.71300	3500

Back Tracking

n	peso	tiempopromedio(ms)	ganancia maxima
0	1 3500	0.00000	0
1	2 3500	0.00000	0
2	3 3500	0.00000	0
3	4 3500	0.00000	0
4	5 3500	0.00000	3100
5	6 3500	0.00000	3100
6	7 3500	0.00000	3100
7	8 3500	0.00000	3100
8	9 3500	0.00000	3500
9	10 3500	0.00000	3500
10	11 3500	0.62400	3500
11	12 3500	0.00000	3500
12	13 3500	0.62400	3500
13	14 3500	0.62400	3500
14	15 3500	1.87200	3500
15	16 3500	1.87204	3500
16	17 3500	4.36800	3500
17	18 3500	7.48800	3500
18	19 3500	14.97600	3500
19	20 3500	29.95200	3500

Meet in the middle

n	peso	tiempopromedio(ms)	ganancia maxima
0	1 3500	0.00000	0
1	2 3500	0.00000	0
2	3 3500	0.00000	0
3	4 3500	0.00000	0
4	5 3500	0.00000	3100
5	6 3500	0.00000	3100
6	7 3500	0.00000	3100
7	8 3500	0.00000	3100
8	9 3500	0.62400	3500
9	10 3500	0.00000	3500
10	11 3500	0.00000	3500
11	12 3500	0.62400	3500
12	13 3500	0.62400	3500
13	14 3500	1.24800	3500
14	15 3500	0.62400	3500
15	16 3500	2.49600	3500
16	17 3500	2.49600	3500
17	18 3500	4.36804	3500
18	19 3500	4.99200	3500
19	20 3500	10.60800	3500

Dinamica

n	peso	tiempopromedio(ms)	ganancia maxima
0	1 3500	0.62400	0
1	2 3500	1.24800	0
2	3 3500	0.62400	0
3	4 3500	1.24800	0
4	5 3500	1.24800	3100
5	6 3500	1.24800	3100
6	7 3500	1.24800	3100
7	8 3500	1.87200	3100
8	9 3500	1.87204	3500
9	10 3500	2.49600	3500
10	11 3500	1.87200	3500
11	12 3500	2.49600	3500
12	13 3500	2.49600	3500
13	14 3500	3.12000	3500
14	15 3500	3.12000	3500
15	16 3500	3.12000	3500
16	17 3500	3.74404	3500
17	18 3500	3.74400	3500
18	19 3500	3.74400	3500
19	20 3500	4.36800	3500

Caso 2 tabla:

Fuerza Bruta

	n	peso	tiempopromedio(ms)	ganancia maxima
0	1	3500	0.04000	1
1	2	3500	0.00000	11
2	3	3500	0.04000	111
3	4	3500	0.00000	311
4	5	3500	0.04000	711
5	6	3500	0.12004	1211
6	7	3500	0.24000	1811
7	8	3500	0.40004	2461
8	9	3500	0.32000	3161
9	10	3500	0.64004	3441
10	11	3500	1.52008	3491
11	12	3500	2.68020	3500
12	13	3500	5.24028	3500
13	14	3500	10.52060	3500
14	15	3500	20.80120	3500
15	16	3500	41.28230	3500
16	17	3500	82.40470	3500
17	18	3500	165.77000	3500
18	19	3500	336.17900	3500
19	20	3500	671.31800	3500

Meet In the middle

	n	peso	tiempopromedio(ms)	ganancia maxima
0	1	3500	0.00000	1
1	2	3500	0.04000	11
2	3	3500	0.00000	111
3	4	3500	0.04000	311
4	5	3500	0.00000	711
5	6	3500	0.04000	1211
6	7	3500	0.08000	1811
7	8	3500	0.08000	2461
8	9	3500	0.12004	3161
9	10	3500	0.20000	3441
10	11	3500	0.24000	3491
11	12	3500	0.48004	3500
12	13	3500	0.56004	3500
13	14	3500	1.00004	3500
14	15	3500	1.16008	3500
15	16	3500	2.20012	3500
16	17	3500	2.48016	3500
17	18	3500	4.68024	3500
18	19	3500	5.32032	3500
19	20	3500	10.04060	3500

BackTracking

	n	peso	tiempopromedio(ms)	ganancia maxima
0	1	3500	0.00000	1
1	2	3500	0.04000	11
2	3	3500	0.00000	111
3	4	3500	0.04000	311
4	5	3500	0.00000	711
5	6	3500	0.04004	1211
6	7	3500	0.08000	1811
7	8	3500	0.08000	2461
8	9	3500	0.12000	3161
9	10	3500	0.16000	3441
10	11	3500	0.28004	3491
11	12	3500	0.40000	3500
12	13	3500	0.64004	3500
13	14	3500	1.00004	3500
14	15	3500	1.28008	3500
15	16	3500	1.32008	3500
16	17	3500	1.32008	3500
17	18	3500	1.32008	3500
18	19	3500	1.32008	3500
19	20	3500	1.40008	3500

Dinamica

	n	peso	tiempopromedio(ms)	ganancia maxima
0	1	3500	1.00004	1
1	2	3500	1.68012	11
2	3	3500	1.52008	111
3	4	3500	1.80012	311
4	5	3500	2.08008	711
5	6	3500	2.32016	1211
6	7	3500	2.56012	1811
7	8	3500	2.76016	2461
8	9	3500	2.96020	3161
9	10	3500	3.20016	3441
10	11	3500	3.36020	3491
11	12	3500	3.60020	3500
12	13	3500	3.80024	3500
13	14	3500	4.00020	3500
14	15	3500	4.12024	3500
15	16	3500	4.24024	3500
16	17	3500	4.32024	3500
17	18	3500	4.40028	3500
18	19	3500	4.48024	3500
19	20	3500	4.56028	3500

Bibliografia:

http://www.numeroalazar.com.ar/

https://www.random.org/integers/?num=100&min=1&max=1000000&col=2&base=10&format=html&rnd=new

https://en.wikipedia.org/wiki/Knapsack_problem

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to algorithms. MIT Press, Cambridge, MA, third edition, 2009.