DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Eligiendo justito

Algoritmos y Estructuras de Datos III

Segundo Cuatrimestre de 2018

Integrante	LU	Correo electrónico
Facundo Linlaud	561/16	facundolinlaud@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

1. Análisis del problema

En este documento se analizará el problema de la suma de subconjuntos (*subset sum* en inglés). Dado un conjunto de n elementos V generalizados v_i y un valor objetivo T, decidir si existe un subconjunto de S cuyos elementos sumen exactamente T y, de existir múltiples soluciones, dar el mínimo cardinal. En el futuro, nos referiremos a las instancias de este problema con una tupla de la forma: (V, n, T).

Es interesante analizar este problema y sus derivados por su importante papel, por ejemplo, en las Ciencias de la Computación. Uno de sus caso de uso es encontrar la mejor distribución de tareas a ejecutar en dos procesadores, minimizando tiempos de *idling* no es mas que una instancia del *subset sum* con $T = \frac{total}{2}$, siendo $total$ la suma de los tiempos totales de todas las tareas a ejecutar.

2. Técnicas propuestas

Se nos pidió implementar tres soluciones para el problema utilizando las siguientes técnicas algorítmicas por separado:

- Brute-forcing
- Backtracking
- Programación dinámica

2.1. Brute-forcing y Backtracking

La técnica de Brute Forcing consiste en probar absolutamente todas las combinaciones posibles dado un conjunto de posibles soluciones definido. Si bien esta técnica es sencilla de implementar y asegura encontrar una solución al problema si es que existe, su complejidad suele ser cara y se disponen de alternativas más eficientes que veremos a continuación como Backtracking, que suele ser una solución *inteligente* de fuerza bruta.

2.2. Programación Dinámica

Esta técnica puede ser resumida como *divide, conquer & memoization*, donde se minimizan las llamadas recursivas al extinguir aquellas que ya han sido calculadas anteriormente. Esto sucede cuando existen recursiones que se invocan más de una vez con los mismos parámetros. En este documento, se considerarán dos enfoques de la programación dinámica:

- El enfoque top-down (o *memoization*)
- El enfoque bottom-up (o *tabulation*)

3. Brute-forcing

Sea la tupla (V, n, T) una instancia del problema de la Suma De Subconjuntos, definimos su conjunto de posibles soluciones S (sobre el cual trabajará el algoritmo de *brute force*) como todas las combinaciones de elementos posibles sobre el conjunto V. Es decir, $S = \mathcal{P}(V)$.

3.1. Solución

```
SUBSET-SUM(t)
1  if t = 0 then return 0
2  i = 1
3  while i ≤ n and !Is-Subset-Solution(i, 0, t) do
4       i++
5  if i ≤ n then
6       return i
7  else
8       return ∞
```

```
IS-SUBSET-SOLUTION(k, from, t)
1  if k = 0 then
2       if t = 0 then
3            return true
4       else
5            return false
6  else
7       if k = n − from then
8            return Is-Subset-Solution(k − 1,
             from + 1, t − V[from])
9       else
10           return Is-Subset-Solution(k − 1,
             from + 1, t − V[from]) or
             Is-Subset-Solution(k, from + 1, t)
```

Se puede observar que el algoritmo está dividido en dos partes:

- La función $Subset - Sum$ que toma un parámetro t que representa la suma objetivo (y original) a alcanzar
- La función $Is - Subset - Solution$ que toma tres parámetros: k que representa el tamaño de grupos en los cuales se intentará agrupar el conjunto de valores originales para llegar a t. El segundo parámetro es $from$, un índice que apunta a un elemento dentro de la lista de valores V consideraba global, válida y disponible (junto con su respectivo tamaño n). El tercer parámetro es t y lleva la suma acumulada objetivo del subproblema.

El funcionamiento del algoritmo es sencillo: se empieza invocando la función $Subset - Sum$ con una suma objetivo determinada. Esta función, a su vez, invoca n veces la rutina $Is - Subset - Solution(i,$ $from = 0$, t = $t)$, con $i = 1, ..., n$. Además, el bucle se seguirá ejecutando mientras esta última rutina no retorne $true$.

¿Qué hace $Is - Subset - Solution$? La función de esta rutina es buscar recursivamente subconjuntos de V con cardinal k donde la totalidad de sus elementos sumen t. Dado la naturalidad recursiva del método, se presentan los siguientes dos casos bases que se dan cuando $k = 0$, es decir, cuando el subconjunto que queremos armar ya tiene la máxima capacidad de elementos que es la que se quiere analizar:

- Si $t = 0$, es decir, si ya sumamos lo que teníamos que sumar, entonces se retorna $true$ dado que el conjunto V tiene un subconjunto de k elementos que sumen t.
- Si $t \neq 0$ significa que todavía no llegamos a t, pero tampoco nos queda espacio para agregar elementos de V porque recordemos que $k = 0$. Luego, se retorna $false$ porque no podemos llegar a t sin sumar ningún elemento.

Luego, los pasos recursivos se dan cuando $k \neq 0$, es decir, cuando todavía tenemos espacio para considerar (o no) un elemento de V en nuestro subconjunto solución:

- Si $k = n - from$ significa que todavía nos quedan elegir k elementos de V y, además, nos quedan k elementos por examinar de V pues el resto ya los agregamos o descartamos. Esto significa que sí o sí tengo que agregar el número actual al que apunta $from$ a la posible solución, sino voy a llegar al final de la lista V y me van a faltar elementos por agregar al subconjunto de k elementos. Luego, retorno el resultado de la recursión incluyendo este número, es decir, incrementándo en 1 a $from$ para analizar el siguiente elemento, decrementándo en 1 el valor de k (porque acabamos de sumar un elemento, por lo tanto nos faltan $k - 1$) y restando V_{from} al valor de t.

- Si $k \neq n - from$ entonces nos podemos dar la libertad de incluir o no al valor V_{from} en nuestra posible solución. Luego, hacemos recursión sobre ambos caminos y, como la rutina es booleana, devolvemos $true$ si cualquiera de los dos es también $true$.

Finalmente, si $Is - Subset - Solution$ devuelve $true$ para algun cardinal k entonces $Subset - Sum$ detendrá el bucle y devolverá el cardinal que resultó tener una solución. Este cardinal es mínimo porque el bucle comienza desde el menor cardinal posible para un subconjunto de V e incrementa secuencialmente hasta llegar a n. Si no se encuentra un subconjunto solución para todos los valores válidos de i, se retorna infinito.

De esta manera, podemos asegurar que el algoritmo de *brute force* llega a una solución, si es que la hay, pues el conjunto de posibles soluciones que son inspeccionadas por el algoritmo abarca todas las combinaciones posibles del conjunto de valores V.

3.2. Complejidad

Es fácil chequear que cada llamada a $Is - Subset - Solution$ se realiza n veces y que el resto de las instrucciones involucradas se computa en $\mathcal{O}(1)$. Luego, basta calcular la complejidad de esta última subrutina para determinar la complejidad del algoritmo entero.

Podemos demostrar que la complejidad de $Is - Subset - Solution$ pertenece a la clase $\mathcal{O}(2^n)$ pues cada llamada a esta función debe resolver reccurentemente, como mucho, 2^n problemas; es decir, para cada elemento e de V, analizar el camino donde V_e pertenece a la posible solución así como el camino donde V_e no pertenece a la posible solución. De hecho, la complejidad real del algoritmo nunca será igual a $n * 2^n$ porque cada recursión se detendrá al haber completado todos los grupos de k elementos. Finalmente, la complejidad final del algoritmo $Subset - Sum$ es $\mathcal{O}(n * 2^n)$.

4. Backtracking

La técnica de backtracking hace foco en el descarte de ramificaciones infactibles dado un problema y ciertas condiciones. A continuación, analizamos la solución al problema de la suma de subconjuntos con backtracking.

4.1. Solución

```
SUBSET-SUM(i, t, cardinal_so_far)
 1  if cardinal_so_far ≥ minimum_cardinal_so_far then
 2      return ∞

 3  if i = 0 then
 4      if t = 0 then
 5          if cardinal_so_far < minimum_cardinal_so_far then
 6              minimum_cardinal_so_far ← cardinal_so_far

 7          return 0
 8      else
 9          return ∞
10  else if values[i] > t then
11      return Subset-Sum(i − 1, t, cardinal_so_far)
12  else
13      return min(Subset-Sum(i − 1, t, cardinal_so_far),
                   1 + Subset-Sum(i − 1, t − values[i], cardinal_so_far + 1))
```

En este algoritmo se asumen globales las variables $values$ (los valores disponibles para intentar sumar T), su respectiva dimensión n y una variable $minimum\_cardinal\_so\_far$ inicializada en infinito. La función $Subset - Sum$ es invocada por primera vez con la tupla $(i, t, cardinal\_so\_far) = (n, T, 0)$.

El funcionamiento del algoritmo es similar al de *brute-force* pero más inteligente. Es decir, en el momento en el que una rama deja de ser factible, esta es descartada inmediatamente. Esto se puede observar en las líneas 10 y 11 donde, si el valor que se está evaluando en el momento excede la suma buscada, entonces se desecha la alternativa. Esto representa una poda por factibilidad y cualquier conjunto de valores tomados como posible solución que excedan T no pueden ser solución. Además, se lleva un registro de cual fue el mínimo cardinal entre todas las soluciones factibles encontradas hasta el momento y se utiliza como límite para futuras exploraciones. Es decir, si se encontró una solución con cardinal m, cualquier subconjunto de $values$ que sume T pero disponga de un cardinal mayor a m no será una solución óptima al problema. Esto se conoce como una poda por optimalidad y se puede observar en las líneas 1, 2, 5 y 6.

Supongamos $S =< s_1, ..., s_k >$ una solución óptima con cardinal finito para un problema (V, n, T). Ya sabemos que el algoritmo encontrará una solución si la hay. Falta analizar qué sucederá cuando la encuentre:

I) S es la primera solución encontrada por el algoritmo, por lo tanto $cardinal\_so\_far$ es ∞. Luego, $|S| < cardinal\_so\_far$ por lo tanto S es persistida como solución óptima.

II) S no es la primera solución encontrada por el algoritmo, por lo tanto $cardinal\_so\_far < \infty$.

   a) Si $|S| < cardinal\_so\_far$ entonces S será mejor solución que la anterior encontrada. Luego, se persiste el cardinal de S como solución óptima.

   b) Si $|S| = cardinal\_so\_far$ entonces S será igual solución que la anterior encontrada. El cardinal de S sigue siendo solución óptima por más que haya sido encontrado a partir de otro conjunto de valores.

   c) Si $|S| > cardinal\_so\_far$ implica que se llegó al valor T sumando menos elementos que S, contradiciendo la hipótesis de que el cardinal de S es solución óptima. Absurdo.

De esta manera, podemos corroborar que la solución real del problema nunca será podada en favor de una peor solución.

4.2. Complejidad

Este algoritmo comparte la misma complejidad de peor caso que la implementación *brute force*. En el peor de los escenarios, el algoritmo debe resolver $n * 2^n$ subproblemas pues por cada elemento a evaluar de $values$, se pueden llegar a computar las 2^n configuraciones de elementos posibles.

5. Programación dinámica

Antes que nada, debemos demostrar que el Principio de Optimalidad de Bellman vale para el problema de la suma de subconjuntos. Es decir, que toda solución óptima de un problema es construida a partir de subsoluciones óptimas de sus respectivos subproblemas. ¿Entonces, se cumple el principio de optimalidad? Verifiquémoslo.

5.1. Cumplimiento del Principio de Optimalidad

Asumiendo I un conjunto solución de n índices correspondientes a elementos en V tal que $I = \{1, ..., n\}$ y $\sum_{i \in I} v_i = T$ con T el valor objetivo. Para cada valor posible en V hay dos posibilidades: que sea parte de la solución o no lo sea.

I) $n \notin I \implies I \subseteq \{1, 2, ..., n-1\}$

   De esta manera, el valor de T no cambia. Es decir: $\sum_{i \in I} v_i = T$ con I solución óptima por hipótesis. Luego, I remanece solución óptima para el subproblema de valor objetivo T.

II) $n \in I$:

En este caso, se tiene que cumplir $\sum_{i \in I'} v_i = T - v_i$ con $I' = I - \{n\}$ y I' solución óptima.

Si $I' = I - \{n\}$ **no** fuese solución óptima entonces existiría un $I = I''$ mínimo (y óptimo) tal que $|I''| < |I'|$ y $\sum_{i \in I''} v_i = T - v_n$

Pero si I'' fuese solución óptima del subproblema $n-1$ entonces la solución óptima del problema n sería $I = I'' \cup \{n\}$ y recordando la equivalencia $I = I' \cup \{n\}$ que fue obtenida anteriormente, podemos generar un sistema de ecuaciones y luego aplicar álgebra:

$$\begin{cases} I' = I - \{n\} \\ I = I'' \cup \{n\} \end{cases} \iff I' = (I'' \cup \{n\}) - \{n\} \iff I' = I''$$

La equivalencia entre I' e I'' obtenida como conclusión contradice la declaración $|I''| < |I'|$, obteniéndose un absurdo. Luego, I' es subsolución óptima del subproblema.

Finalmente, el problema satisface el principio de optimalidad. □

De esta manera, formulamos el problema recursivamente:

$$f(i,t) = \begin{cases} 0 & \text{si } i = 0 \wedge t = 0 \\ \infty & \text{si } i = 0 \wedge t > 0 \\ min\{f(i-1,t), f(i-1,t-v_i)+1\} & \text{sino} \end{cases}$$

5.2. Solución

```
SUBSET-SUM(i, accumulator)
 1  if (i, accumulator) ∈ dic then                                   // O(1)
 2      if i = 0 then                                                // O(1)
 3          if accumulator = 0 then                                  // O(1)
 4              dic[i][accumulator] ← 0                              // O(1)
 5          else
 6              dic[i][accumulator] ← ∞                              // O(1)
 7      else
 8          if V[i] > T then                                         // O(1)
 9              dic[i][accumulator] ← Subset-Sum(i − 1, accumulator)
10          else
11              dic[i][accumulator] ← min(Subset-Sum(i − 1, accumulator),
12                  1 + Subset-Sum(i − 1, accumulator − V[i]))
13      return dic[i][accumulator]                                   // O(1)
```

Este algorítmo tiene un enfoque top-down. Es invocado con los parámetros $(i, accumulator)$ donde i representa el i-ésimo elemento dentro de un conjunto V de valores posibles utilizados para intentar sumar el valor objetivo T. La línea 1 pregunta si la dupla $(i, accumulator)$ está *cacheada*, es decir, si ya fue calculada anteriormente. Si la respuesta es no, se procede a calcularlo. El siguiente condicional representa una de las condiciones necesarias para el caso base: que nos hayamos quedado sin elementos para analizar en el conjunto V. Si esto sucede, el algoritmo encontró una respuesta para este camino recursivo, pero depende del valor de $accumulator$:

- Si $accumulator$ es cero, entonces la solución del camino recursivo es cero. Es decir, necesito 0 elementos del conjunto V para sumar 0.

- En cambio, si el valor de $accumulator$ no es cero, el camino no tiene solución, pues el camino recursivo no dispone de elementos suficientes para sumar $accumulator$.

Si el valor de i es distinto de cero, entonces estamos en presencia de un paso recursivo. Aquí existen dos casos posibles:

- $V[i]$ es mayor al valor acumulado al que se quiere llegar; es decir, el elemento del conjunto V que se está analizando supera el valor de $accumulated$ por lo tanto no puede ser parte de la solución. En este, caso el algoritmo continúa por el camino recursivo donde $V[i]$ no es solución.
- $V[i]$ no es mayor al valor acumulado y podría llegar a ser parte de la solución. En este caso, se consideran y exploran ambos caminos: donde $V[i]$ es incluído en la solución y donde no lo es. El algoritmo se quedará con el menor de los dos caminos. Es decir, el de menor cardinalidad. Aquí es importante aclarar la importancia de devolver ∞ en el caso base sin solución explicado anteriormente.

Cualquiera sea el resultado del camino recursivo, este se guardará en el diccionario global en los índices $(i, accumulator)$ para ser reutilizado la próxima vez que la función $Subset - Sum$ sea invocada con los mismos parámetros.

5.3. Complejidad

Debido a la naturaleza de la técnica de programación dinámica, la complejidad del algoritmo queda definida de la siguiente manera: $\mathcal{O}(f) = \mathcal{O}(\#subproblemas) * \mathcal{O}(\#costo\ por\ subproblema)$.

Para esto, debemos resolver la complejidad del algoritmo exceptuando las recursiones. Es fácil observar que, asumiendo un diccionario con escritura y lectura en $\mathcal{O}(1)$, todo el algoritmo es $\mathcal{O}(1)$. Luego, la complejidad de la solución entera estará determinada por la cantidad de subproblemas totales, que es exactamente $n * T$, recordando que n es la cantidad de números utilizables para llegar al objetivo T. Es decir, el algoritmo sólo hace recursión sobre los subproblemas cuyo resultado no calculó previamente. En caso de haberlo calculado, este estará disponible en el diccionario en $\mathcal{O}(1)$.

6. Análisis de performance

6.1. Aclaraciones

Para generar los siguientes análisis de performance, se creó un programa en *Python* para generar una serie de problemas a computar, con y sin solución, desde 10 hasta n elementos. Se pueden describir los problemas como elementos de una matriz $Problema\ problemas[n][r]$ donde $Problema$ es una tupla del tipo $(T, values)$, n es la cantidad máxima de elementos a evaluar y r es la cantidad de repeticiones para cada problema de tamaño n. Es decir, la cantidad de veces que se va a repetir un experimento de n elementos con el objetivo de mejorar las mediciones obtenidas. Es necesario aclarar que para toda repetición de un problema dado un n, sus valores T serán iguales. Esto es necesario para poder estimar confiablemente la equivalencia de un cómputo $\mathcal{O}(1)$ en segundos. Esto se realiza dividiendo el valor de T contra la cantidad de recursiones que demandó cada problema. De esta manera, se obtiene una cota $\mathcal{O}(n * T)$ aproximada en los análisis de programación dinámica.

Además, cada problema es generado aleatoriamente con un T entre $(99, 9999)$. La mitad de los problemas tienen garantizada una solución. La otra mitad, no. La disposición de los elementos de los valores de $values$ con solución son equiprobables. Es decir, tienen los elementos que forman parte de la solución pueden estar en cualquier lugar de la lista. Es por eso que en algunos gráficos se podrá observar que ciertos problemas de n elementos hayan sido solucionados más rápidamente que otros problemas de k elementos, con $n > k$. Estos casos intentan ser apaciguados mediante la elevada repetición de los problemas para cada cardinal y luego a través de la "poda"del $5\,\%$–$10\,\%$ de las soluciones más rápidas para cada tamaño de problema. Es importante aclarar que todos los algoritmos resuelven exactamente los mismos problemas, asegurando justicia e integridad en las comparaciones.

Específicamente, los experimentos fueron computados para $n = 10..,34$ con $r = 40$. Es decir, cada n fue calculado 40 veces con el mismo T pero distintas listas de valores de n elementos.

6.2. Fuerza Bruta

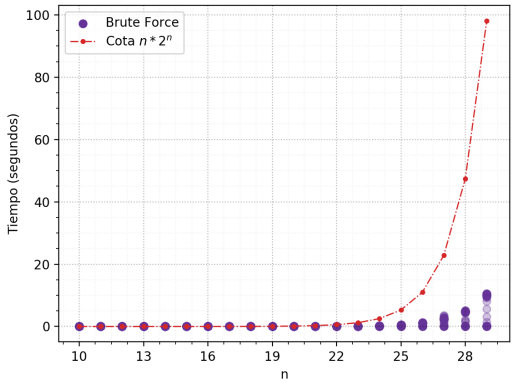

Figura 1: Segundos consumidos (en escala lineal) en función de la cantidad de elementos de $values$.

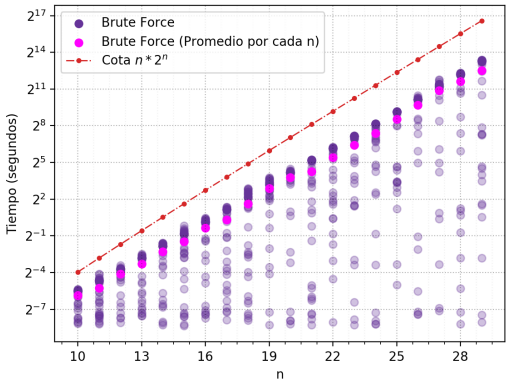

Figura 2: Segundos consumidos (en escala logarítmica) en función de la cantidad de elementos de $values$.

En estos gráficos se puede observar – en púrpura – todas las computaciones para cada problema de cardinal n. Mientras que, en rojo, se puede apreciar la cota teórica del algoritmo de fuerza bruta. Ambos experimentos apoyan empíricamente la hipótesis "*El algoritmo de fuerza bruta tiene complejidad* $\mathcal{O}(n * 2^n)$". Además, queda claro el comportamiento exponencial de la solución. Esta hipótesis puede ser reforzada con el siguiente experimento:

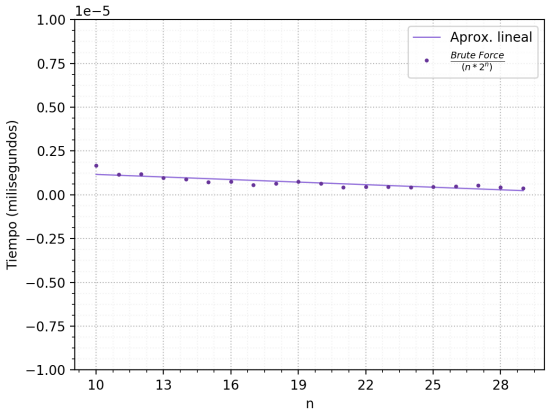

Figura 3: Segundos consumidos en función de la cantidad de elementos de $values$ dividido $n * 2^n$

De esta manera, queda explicitada la tendencia de la complejidad del algoritmo con respecto a su complejidad: siempre será ampliamente menor.

6.3. Back Tracking

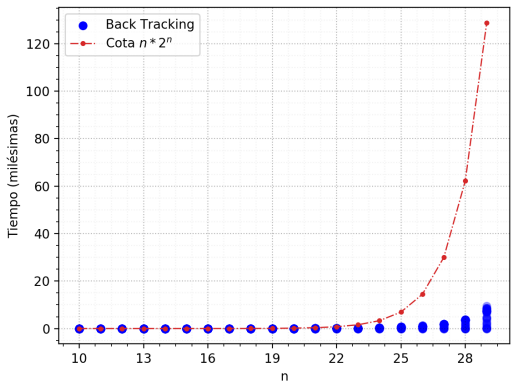

Figura 4: Segundos consumidos (en escala lineal) en función de la cantidad de elementos de $values$.

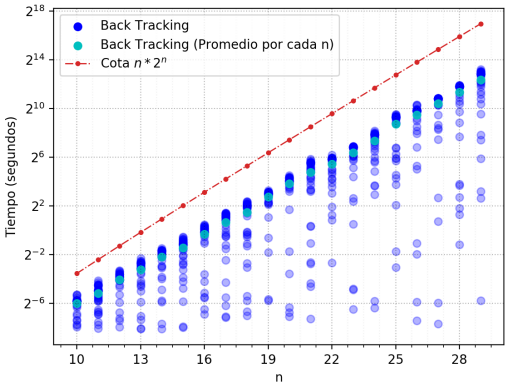

Figura 5: Segundos consumidos (en escala logarítmica) en función de la cantidad de elementos de $values$.

Al igual que en los gráficos de fuerza bruta, la tendencia es clara: la función $f(n) = n * 2^n$ acota superiormente al algoritmo de back tracking para todo $n \in \{10, ..., 34\}$. A su vez, queda evidenciado su comportamiento exponencial.

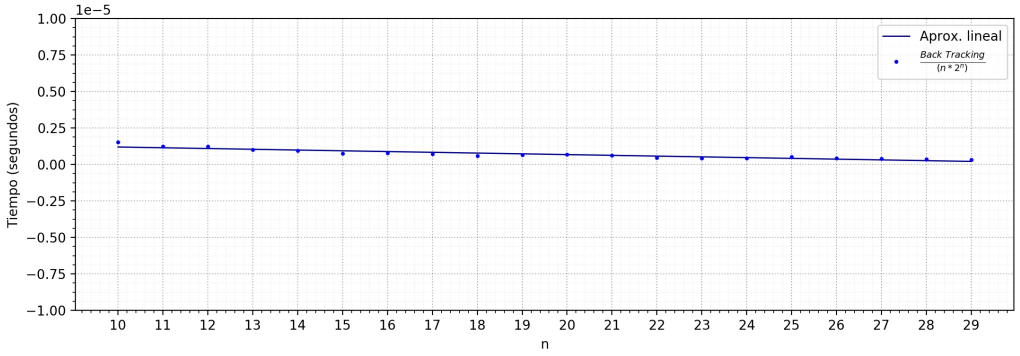

Figura 6: Segundos consumidos en función de la cantidad de elementos de $values$ dividido $n * 2^n$

6.4. Programación Dinámica

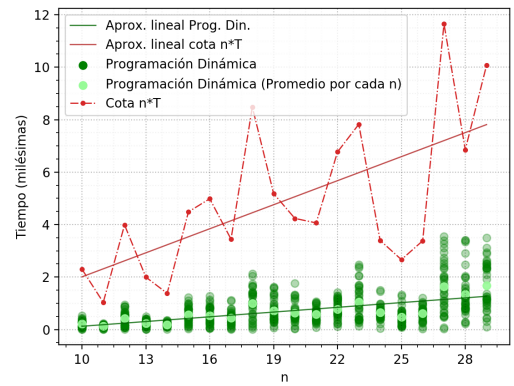

Figura 7: Segundos consumidos (en escala lineal) en función de la cantidad de elementos de $values$.

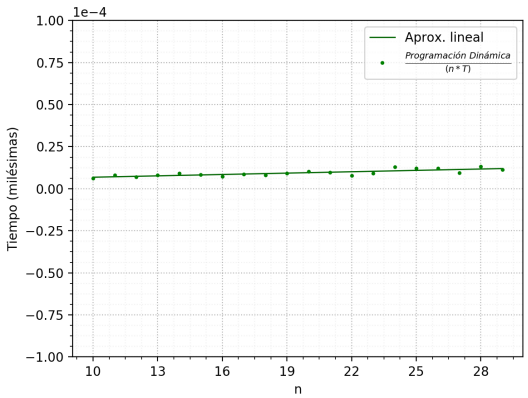

Figura 8: Segundos consumidos en función de la cantidad de elementos de $values$ dividido $n * T$

En estos experimentos podemos notar una tendencia extraña en el grafico 7, donde se aprecia una fluctuación alternante en la cota $n * T$. Al evaluar con detenimiento la situación, podemos observar que las depresiones en la cota también se ven reflejadas en las mediciones del algoritmo de programación dinámica. Lo mismo sucede con las elevaciones. Este fenómeno puede ser explicado sencillamente: la cota teórica del algoritmo es $n * T$ que no tiene una traducción directa con los segundos que puede demorar un algoritmo. Para esto, fue necesario calcular la equivalencia en segundos que llamaremos δ de una operación $\mathcal{O}(1)$ en nuestro algoritmo. Luego, estimamos el costo temporal en segundos de un problema (V, n, T) como $n * T * \delta$. Si bien δ permanece igual para todos los n (porque se utilizan todos los cardinales para calcularlo), T no permanece inmutable. Es por eso que el valor final de $n * T * \delta$ es afectado por un valor T que es elegido al azar para cada cardinal, explicando el por qué de las depresiones y elevaciones en la cota pero que, a su vez, son respetadas por el algoritmo de programación dinámica.

6.5. Algoritmo vs algoritmo

6.5.1. Entradas densas y aleatorias

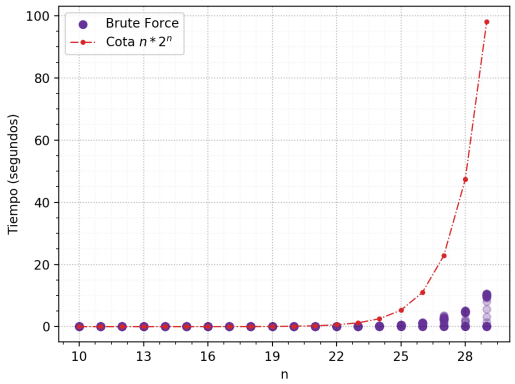

Figura 9: Segundos consumidos en función de la cantidad de elementos de $values$

Podemos observar la superioridad en performance de la implementación con Programación Dinámica versus Back Tracking y Fuerza Bruta para entradas densas y aleatorias, es decir, con una cantidad alta de elementos. Mientras que la primera es completamente lineal, las dos últimas son exponenciales. Pero este caso se da siempre y cuando la composición de los problemas sea totalmente aleatoria. ¿Qué situación tendríamos si las soluciones se encontrasen en una posición temprana de la lista? ¿Y si el valor objetivo T es muy grande? ¿Y si además las listas fuesen pequeñas?

6.5.2. Soluciones tempranas o tardías con T elevado

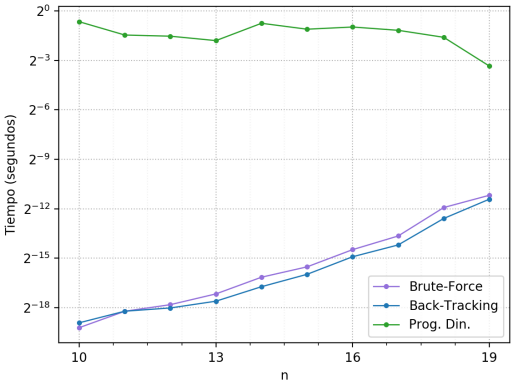

Figura 10: Problemas sólo con soluciones tempranas (es decir, exclusivamente en el principio de $values$)

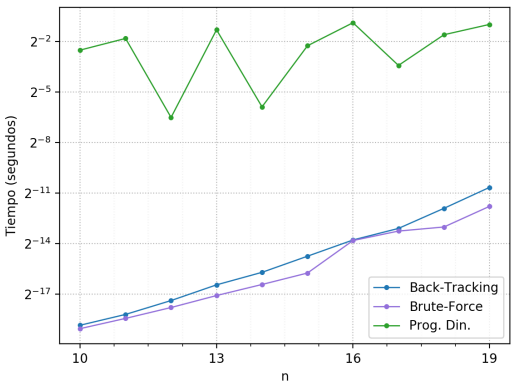

Figura 11: Problemas sólo con soluciones tardías (es decir, exclusivamente en el final de $values$)

Para este experimento utilizamos un valor T de ocho dígitos decimales. Podemos observar que en ambos casos el algoritmo de Programación Dinámica no tiene un buen desempeño. En cambio, el de Fuerza Bruta y Back Tracking compiten cercanamente. Cuando la solución a los problemas está exactamente

al principio de la lista, el algoritmo de fuerza bruta resulta más eficiente y porque este – por como fue implementado – inicia su búsqueda desde el comienzo de la lista. En cambio, cuando la solución está al final de la secuencia, el algoritmo de Back Tracking es ligeramente más rápido por el motivo análogo al anterior: Back Tracking inicia su búsqueda desde el final de la lista de valores. ¿Pero qué pasa con el algoritmo de Programación Dinámica? No parece ser eficiente en ningún aspecto. Hipotetizamos que es por el excesivamente elevado valor de T. Analizaremos este aspecto en la siguiente sección.

6.5.3. Cardinal fijo y valor objetivo dinámico con listas pequeñas

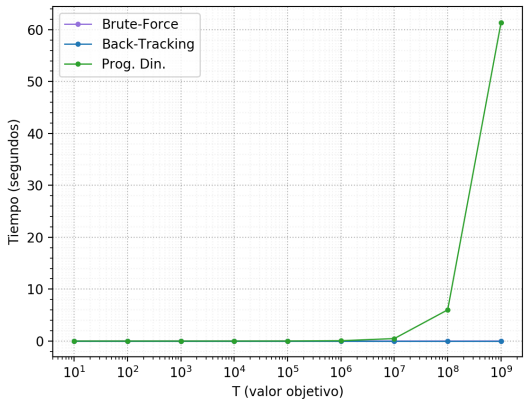

Figura 12: Segundos consumidos en función del valor objetivo T

En este experimento fijamos el cardinal de los valores posibles a $n = 5$ e iteramos sobre el valor de T. Podemos ver que para los algoritmos de Back Tracking y Brute Force, solucionar un problema de tamaño 5 para cualquier T es instantáneo mientras que los tiempos tomados por la implementación de Programación Dinámica aumentan drásticamente a medida que T lo hace.

7. Conclusiones

Podemos concluir, finalmente, que si bien todas las cotas teóricas son conocidas, el desempeño de ellos depende explícitamente de las características de los datos de entrada. Por ejemplo, la implementación que utiliza la técnica de Programación Dinámica parecía ser la más eficiente de las tres hasta que comenzaron a aparecer valores objetivos muy grandes y las técnicas de Fuerza Bruza y Back Tracking comenzaron a ganar terreno frente a la primera en cuestión. A la hora de evaluar una implementación frente a la otra, es necesario conocer exactamente qué tipo de entradas se desea procesar y cuál es su formato. Es posible, además, considerar mejoras en los algoritmos propuestos. Por ejemplo, uno podría partir en dos el valor de T (por ende la complejidad en el algoritmo de Programación Dinámica se reduciría a la mitad) y buscar la suma de una de sus partes dentro de la lista siempre y cuando en el resto de los elementos no tomados también haya una suma de esta mitad. Esta alternativa es una instancia del Problema de la Suma de Subconjuntos y es un tópico que merece ser estudiado en profundidad.