DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Domingo 16 de Septiembre de 2018 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Kennedy Williams Rios Cuba	381/15	wrios@dc.uba.ar

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

Índice

1. Introducción al Problema

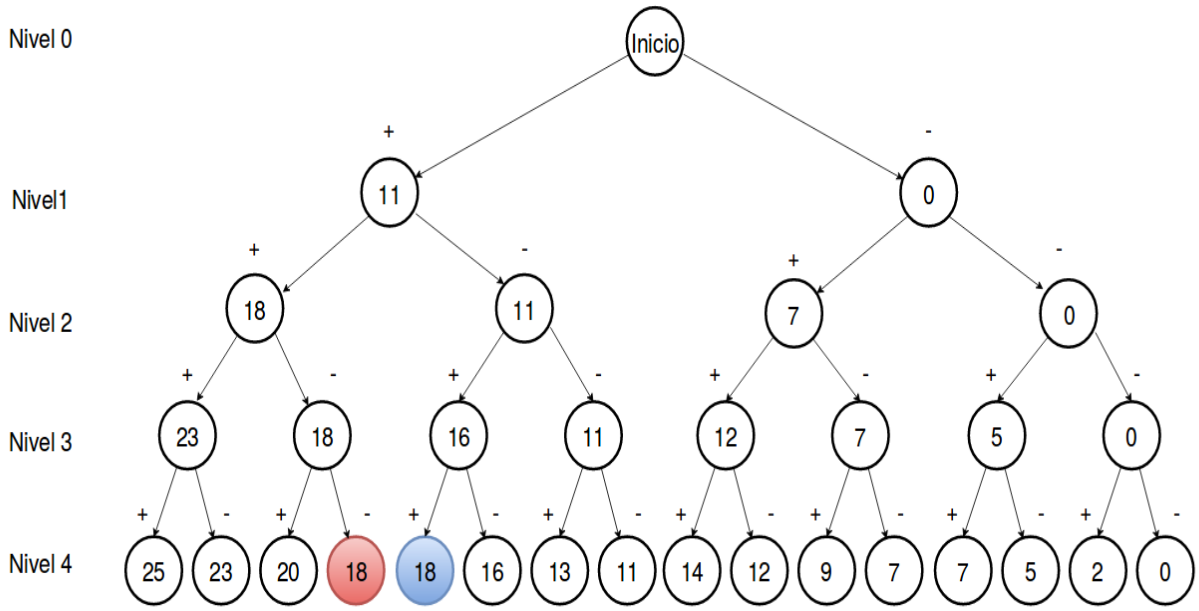
1.1. Definición del problema Subset Sum

Dado un conjunto de n ítems S, cada uno con un *valor* asociado v_i , y un valor objetivo V , decidir si existe un subconjunto de ítems de S que sumen exacto el valor objetivo V . Si existe dicho conjunto, decir cual es la mínima cardinalidad P entre todos los subconjuntos posibles.
En otras palabras decidir si existe$R \subseteq S$ tal que $\sum_{i \in R} v_i = V$, y si existe, devolver la menor cardinalidad posible de R.
Para este problema, asumiremos que todos los valores mencionados son enteros no negativos.

1.2. Ejemplo de problema

1.2.1. Ejemplo 1

En este ejemplo se puede ver cual es el espacio de soluciones, la forma de recorrerlo será definida más adelante de acuerdo a cada algoritmo que usemos para resolver el problema. Además se verá que tamaño tiene este espacio, y la cantidad de posibles soluciones (en el diagrama se logra ver como cada nodo hoja es una posible solución que llego desde las decisiones tomadas).
Se considera un conjunto de 4 elementos $\{11, 7, 5, 2\}$.
Se busca que la suma de los elementos sea 18 y que el cardinal sea mínimo.
El nivel i representa la decisión de tomar el elemento (con un + arriba del nodo) o no tomar el elemento (con un - arriba del nodo) para mirar si forma parte de la solución. Como se puede ver en el gráfico, hay dos soluciones $\{11, 7\}$ y $\{11, 5, 2\}$ que suman 18 pero la mejor es usando menos elementos, con lo cual nos quedamos con la solución $\{11, 7\}$.



1.2.2. Ejemplo 2

Conjunto $\{2, 3, 12, 14, 4\}$, buscamos un subconjunto de elementos que sume 13.
El 14, no puede sumar 13. El 12 no se puede elegir porque no hay elementos que sumen 1. Al combinar

cualquiera de los que restan, no alcanzan a sumar 13 con lo cual no hay solución para este problema.

1.3. El objetivo del trabajo práctico

Resolver el problema propuesto de diferentes maneras realizando posteriormente una comparación entre los diferentes algoritmos utilizados.

2. Desarrollo Problema

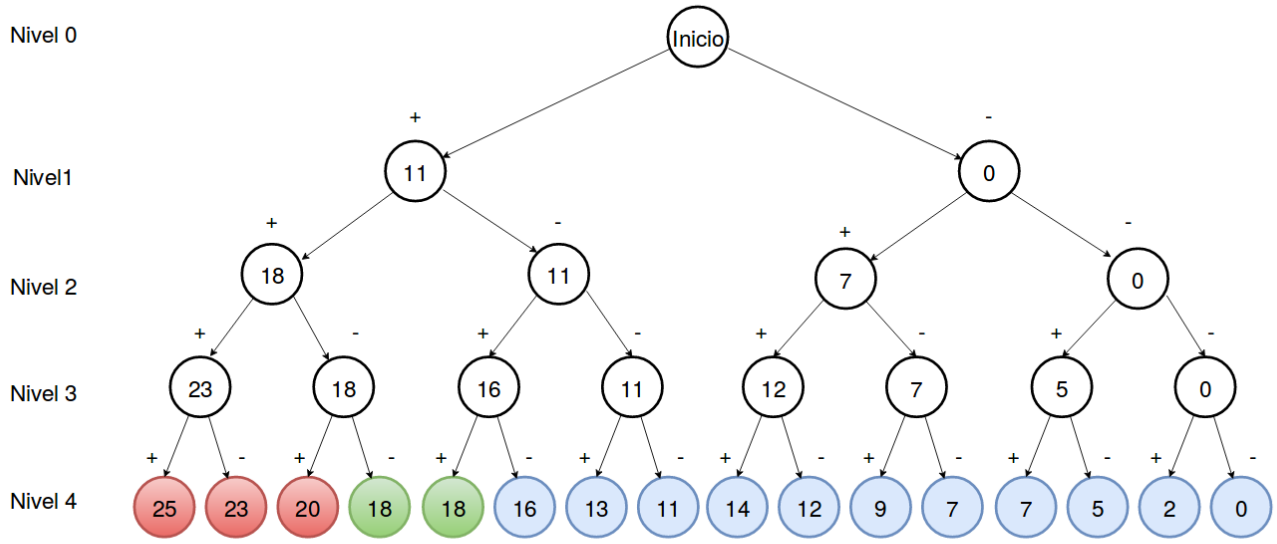
2.1. Caracterización de una solución

Una solución de nuestro problema es un subconjunto de elementos del conjunto original tal que la suma de ellos es exactamente el valor objetivo.

2.2. Espacio de soluciones

Se redefine Solución para poder hablar de factibilidad y optimalidad.
En la imagen podemos ver, que luego de diferentes decisiones, llegamos a diferentes soluciones (hojas) y están separados en tres casos.

- Los nodos rojos, verdes y celestes son las soluciones.
- Los nodos verdes son las soluciones factibles.
- Los nodos rojos son las soluciones no factibles.



2.3. Recorrido del espacio de soluciones

Se verán diferentes formas de pensar el problema, y en base a ello elegiremos una forma de obtener la solución.

- La primer forma de ver el problema es simple, mirar todo el espacio de soluciones y quedarnos con la mejor.

- En la segunda forma trataremos de recorrer solamente el espacio de soluciones factibles.

- La tercer forma va mas enfocado a cuando una solución es mejor que otra, y veremos como recorrer las soluciones que son mejores que la solución que tenemos hasta el momento (si las hay, sino podaremos).

- La última forma de pensar el problema sera ver los problemas anteriores inmediatos y ver como con ellos se puede construir el problema mas grande.

3. Fuerza Bruta: Recorriendo todo el espacio

3.1. Algoritmo

FB(S, int V, int i, int n) $O(2^n)$

```
1  if i <= n :        O(1)
2        return min(1 + FB(S, V − C_i, i + 1, n), FB(S, V, i + 1, n))
3  if V = 0 :         O(1)
4        return 0
5  return ∞
```

ResolverFB(S, int V, int i, int n) $O(2^n)$

```
1  sol ← FB(S, V, i + 1, n)
2  if sol = ∞ :       O(1)
3        return −1
4  return sol
```

3.2. Correctitud

El algoritmo recorre todo el espacio de busqueda, lo único que hace es guardar la mejor solución hasta el momento, y luego devuelve la solución que tiene o -1 si no encontró solución.
El algoritmo es correcto pues recorre todo el espacio de soluciones y solamente conserva las soluciones válidas y se queda con la mejor.

3.3. Complejidad

El algoritmo se divide en dos casos, donde se llama recursivamente con un elemento menos.
Definiendo la siguiente ecuación de recurrencia:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T(n) + 1 & \text{si } n > 1 \end{cases}$$

Como podemos ver, la recurrencia:
$T(n) = 2T(n-1) + 1 = 2(2T(n-1) + 1) + 1 = ... = 2^{n-1} + (n-1) = O(2^n)$.
Con lo cual tiene complejidad: $O(2^n)$.

4. Backtracking: Fuerza Bruta Inteligente

4.1. Factibilidad: recorrido siempre válido

Una camino en el árbol del espacio de soluciones, es válido, siempre y cuando la suma de sus elementos no exceda el valor objetivo. Con esto en mente, si encontramos un camino para el cual la suma de sus elementos es mayor al valor objetivo, al estar trabajando con enteros positivos, si seguimos agregando elementos la

suma siempre excederá el valor objetivo, por lo tanto cortamos esa rama del espacio de soluciones y seguimos por otra rama.

4.1.1. Algoritmo

FAC(S, *int* V, *int* i, *int* n) O(2^n)

```
1  if i ≤ n :        O(1)
2      if V − S_i > 0 :        O(1)
3          return min(Fac(S, V, i + 1, n), 1 + Fac(S, V − S_i, i + 1, n))
4      if V = 0 :        O(1)
5          return 0
6  return ∞
```

RESOLVERFAC(S, int V, int i, int n) O(2^n)

```
1  sol ← Fac(S, V, i + 1, n)
2  if sol = ∞ :        O(1)
3      return −1
4  return sol
```

4.1.2. Correctitud

El algoritmo recorre solomente las ramas que tienen solución, con lo cual, llega a las mismas soluciones que Fuerza Bruta y lo único que hace es quedarse con la mejor solución.

4.1.3. Complejidad

Vemos que el algoritmo define la misma recurrencia que fuerza bruta:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T(n) + 1 & \text{si } n > 1 \end{cases}$$

Con lo cual tiene complejidad en peor caso: $O(2^n)$.

4.2. Optimalidad: Recorrido siempre óptimo

Para nuestro problema, una solución es óptima, si la cantidad de elementos de la solución es menor o igual que la cantidad de elementos para toda otra solución.
Dado una rama del espacio de búsqueda definido por el recorrido, si la cantidad de elementos de la rama es mayor o igual a la cantidad de elementos de la mejor solución encontrada hasta el momento, entonces se poda esa rama.
La poda solo corta las ramas con mayor o igual cantidad de elementos que pueden o no llegar a una solución

4.2.1. Algoritmo

OP(S, *int* V, *int* i, *int* n, *int* *cantidad*, *int* *optActual*) O(2^n)

```
1  if i ≤ n :        O(1)
2      if cantidad < optActual :
3          optActual ← cantidad
4          return min(Op(S, V, i + 1, n, cantidad, optActual), 1 + Op(S, V − C_i, i + 1, n, cantidad, optActual))
5      if V = 0 :
6          return 0
7  return ∞
```

```
ResolverOp(S, int V, int i, int n)        O(2^n)
1  sol ← Op(S, V, i + 1, n)
2  if sol = ∞ :                           O(1)
3        return −1
4  return sol
```

4.2.2. Correctitud

Supongamos que existe solución y el algoritmo no encuentra, el algoritmo no poda (hace Fuerza Bruta) hasta encontrar una solución, como no encuentra solución Fuerza Bruta no encuentra solución.
Absurdo pues Fuerza Bruta explora todo el espacio de soluciones.
Por lo tanto, si hay solución la encuentra.
Supongamos que hay una solución óptima S de cardinal k y el algoritmo no lo encuentra:
Sea$n = |S'|$ (S' la solución retornada con cardinal n y $n > k$), entonces antes de encontrar a S', no había solución o la anterior solución tenia mayor cardinal.

- Caso Sin Solución anterior: El algoritmo poda toda solución de cardinal $>= n$.
- Caso Había Solución anterior: Sea m el cardinal de la solución anterior, sabemos que $n < m$.
  Se podaron las soluciones $>= m$, en particular no se podo ninguna solución con cardinal $< n$.

Absurdo pues $k < n$ con lo cual S fue podado. Entonces si existe una solución óptima, el algoritmo lo encuentra y lo devuelve.

4.2.3. Complejidad

Vemos que algoritmo define la misma recurrencia que fuerza bruta:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T(n) + 1 & \text{si } n > 1 \end{cases}$$

Con lo cual tiene complejidad en peor caso: $O(2^n)$.

5. Programación Dinámica: Colision y Memoizacion

5.1. Principio del Optimalidad: para Subset Sum

Definimos:

$$F(S, V) = \begin{cases} 0 & \text{si } S = \{\} \quad y \quad V = 0 \\ min(1 + F(S - S_n, V - valor(S_n)), F(S - S_n, V)) & \text{si } n \geq 1 \\ \infty & \text{si } S = \{\} \quad y \quad V < 0 \end{cases} \tag{1}$$

Sea $S1$ la solución óptima usando $X_1, ..., X_{n-1}$, tal que sumen $V - X_n$.
Sea $S2$ la solución óptima usando $X_1, ..., X_{n-1}$, tal que sumen V.
Sea $S = min(S1, S2)$, la solución óptima para el problema de $n-$elementos, entonces hay 2 opciones:
Usar el $n-$esimo elementos, o no usarlo.
Supongamos que S no es solución óptima, como S no es óptimo entonces existe S' tal que $|S'| < |S|$.

- Caso X_n pertenece a S: $|S'| < |S| = |S1| + 1$, entonces $|S'| - 1 < |S1|$, como $S1$ es el óptimo para sumar $V - X_n$ usando $X_1, ..., X_{n-1}$ y $S' - X_n$ es un conjunto de elementos que suman $V - X_n$ y es menor que el óptimo. Absurdo pues $S1$ es óptimo.
- Caso X_n no pertenece a S: $|S'| < |S| = |S2|$. Absurdo pues $S2$ es el óptimo para sumar V usando $X_1, ..., X_{n-1}$.

Con lo cuál, podemos usar Programación dinámica para resolver el problema y F(S,V) es la función que calcula el valor optimo.

5.2. Algoritmo

PD(S,matrix M, int f, int c) O($V * n$)

```
1  if c = 0 :        O(1)
2        return 0
3  if M[f][c] == ∞ :        O(1)
4        M[f][c] ← min(PD(S, M, f − 1, c), 1 + PD(S, M, f − 1, c − S_f))
5  return M[f][c]
```

ResolverPD(S, int V, int i, int n) O($V * n$)

```
1  sol ← PD(S, V, i + 1, n)
2  if sol = ∞ :        O(1)
3        return −1
4  return sol
```

5.3. Correctitud

La demostración de correctitud es directa porque algoritmo calcula exactamente la función, con lo cual el algoritmo es correcto.

5.4. Complejidad

La complejidad es la cantidad de llamadas recursivas, como se puede ver, la función decrece en dos direcciones, llenando la matriz. Y la cantidad de llamados que se hace en peor caso es $V * n$ por que cada subproblema se resuelve una ves y se guarda en la matriz para ser accedido si se vuelve a solicitar.
Por lo tanto la complejidad es $O(V * n)$

6. Experimentación

6.1. Hipótesis

Con los experimentos se busca comprobar las siguientes hipotesis:
1. Hacer un sort de los elementos de mayor a menor, hace que las podas sean más efectivas.
2. Hacer un sort de los elementos de menor a mayor, hace que las podas empeoren.
3. La podas podas empeora a medida que el valor de los elementos se acercan a 0, y mejora a medida que el valor de los elementos se acercan al valor objetivo.
4. Fuerza Bruta depende únicamente de la cantidad de items.
5. Las podas de factibilidad y optimalidad en mejor caso tienen complejidad lineal.

6.2. Consideraciones para las experimentaciones

6.2.1. Generación de elementos

- Se usa la función sort de c++ (ordena los elementos en forma creciente)
- Los conjuntos fueron generados aleatoriamente, con la función rand de c++ (distribución uniforme) para poder analizar en caso promedio que pasaba con cada algoritmo, en el rango entre $[0..V]$.
- la cantidad de iteraciones es igual a 50, pues se logro ver durante la experimentación que los algoritmos se estabilizaban.
- El rango para la suma fue [15000...800000], porque es limitada por Programación Dinámica, la cual pide memoria de tamaño $n * W$ y al ser W tan grande, hace que no se pueda seguir experimentando.

- El rango para la cantidad de elementos fue [5...35], porque al querer analizar el caso promedio se toma el promedio de 50 iteraciones y los algoritmos exponenciales tardan demasiado.

6.2.2. Imágenes

- La etiqueta *decreciente* indica que los elementos fueron ordenados de menor a mayor para hacer la experimentación.
- La etiqueta *creciente* indica que los elementos fueron ordenados de menor a mayor para hacer la experimentación.
- La etiqueta *unSort* indica que los elementos no fueron ordenados.

6.3. Complejidades Teóricas en la práctica

6.3.1. Fuerza Bruta

La Figura 1 muestra como Fuerza Bruta no es afectada por el sort de los elemento, y al aplicarle el logaritmo en base dos se logra ver que es lineal, lo cual es esperado pues es exponencial en la cantidad de elementos con base igual a 2.

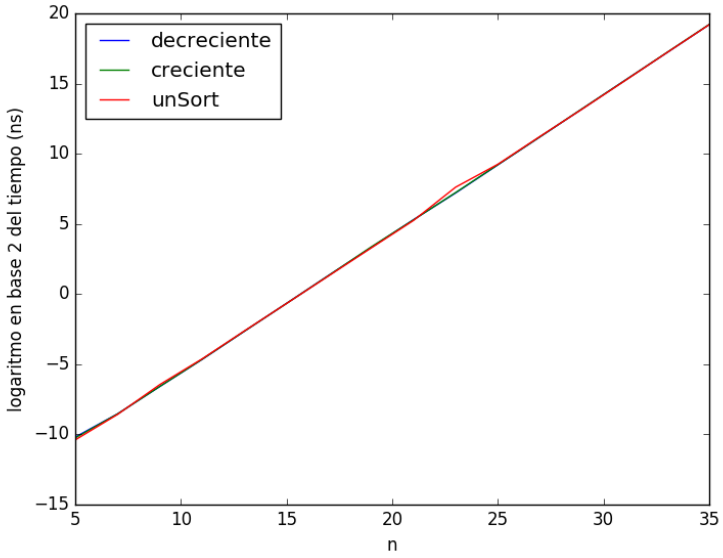


Figura 1: Relación entre el orden de los elementos y la complejidad temporal

6.3.2. Backtracking poda por factibilidad

Tener los elementos ordenados de mayor a menor hace que la poda sea mas efectiva pues al considerar los elementos de mayor valor primero el algoritmo se ahorra comparaciones y empeora cuando el ordenamiento es de menor a mayor.
Por ejemplo: $Conj1 = \{1, 2\}$, $Conj2 = \{2, 1\}$ y el valor objetivo 2.
Para el $Conj1$, considera los subconjuntos $\{1\}$, $\{1, 2\}$, y $\{2\}$.
Para el $Conj2$, considera los subconjuntos $\{2\}$ pues no necesita agregar nada mas para llegar al valor objetivo y $\{1\}$.
El algoritmo hace una comparación menos, pero si el conjunto es mas grande podría hacer menos comparaciones si la suma de los elementos exceden el valor objetivo.
Se aplica logaritmo en base dos y se logra ver que el resultado es lineal.
Los picos de la función se deben a las podas realizadas.

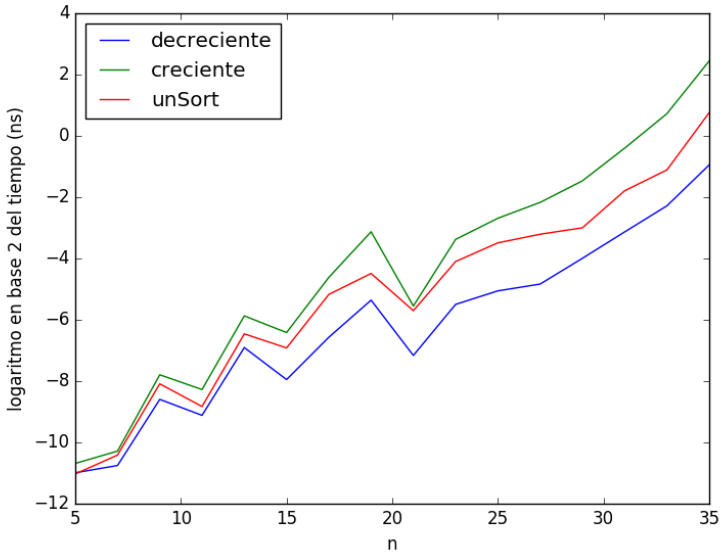


Figura 2: Relación entre el orden de los elementos y la complejidad temporal

6.3.3. Backtracking poda por optimalidad

El ordenamiento de los elementos afecta a la poda de optimalidad.
Por ejemplo: $Conj1 = \{3, 4\}$, $Conj2 = \{4, 3\}$ y el valor objetivo 4.
Para el $Conj1$, considera los subconjuntos $\{3\}$, $\{3, 4\}$, y $\{4\}$.
Para el $Conj2$, considera los subconjuntos $\{4\}$ por que la suma del primer subconjunto es el valor objetivo y el cardinal óptimo es 1, entonces no considera conjuntos mayores o iguales a 1

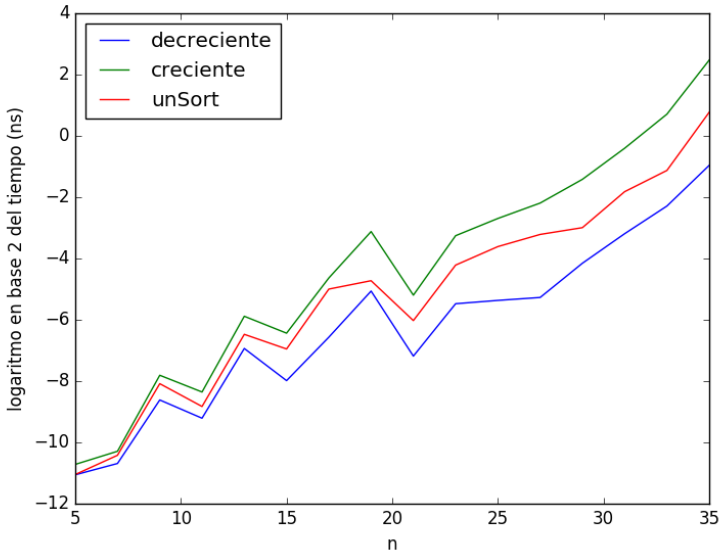


Figura 3: Relación entre el orden de los elementos y la complejidad temporal

6.3.4. Programación dinámica

En este experimento se fija $n = 35$ y se varia V para ver como afecta a la complejidad temporal.
En la figura 4 se ve como Programacion Dinamica es afectado directamente por el valor objetivo y en menor medida por el ordenamiento de los elementos.

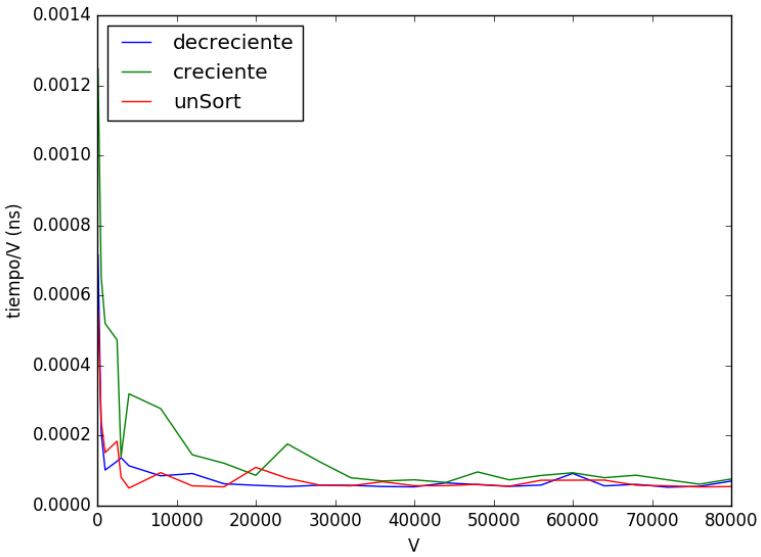


Figura 4: Relación entre el orden de los elementos, el valor objetivo y la complejidad temporal

6.3.5. Influencia del valor de los elementos

Para este experimento se fija $V = 800000$ y $n = 25$.
En la figura 5 el valor de los elementos se genera entre $[0, ..., V/i]$ para la posición i.
Se logra ver que a medida que los elementos decrecen las podas no son tan efectivas.
Fuerza Bruta no es afectado por el valor de los elementos.
Programación dinámica es menos afectada por el decrecimiento del valor de los elementos.

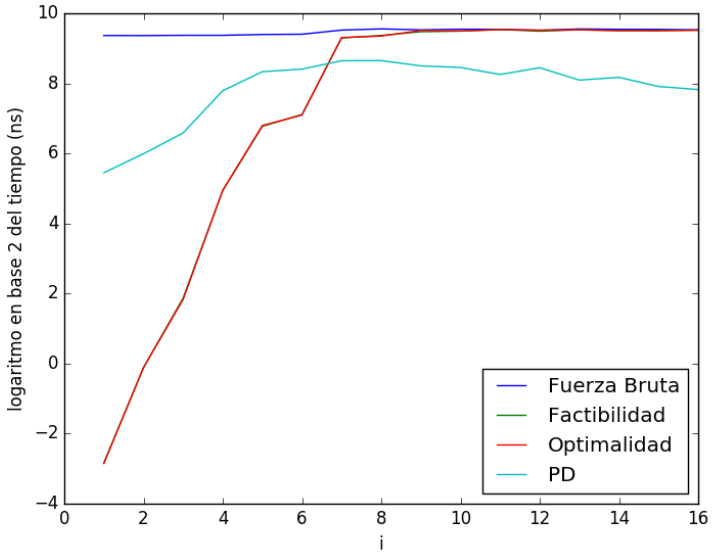


Figura 5: Relación entre el decrecimiento de el valor de los elementos y como afecta a las complejidades

Este caso fue generado para ver el comportamiento lineal de estos algoritmos en mejor caso.
Los elementos son exactamente el valor objetivo y a medida que va creciendo la cantidad de elementos la complejidad temporal dividida por la cantidad de elementos se ajusta mejor.

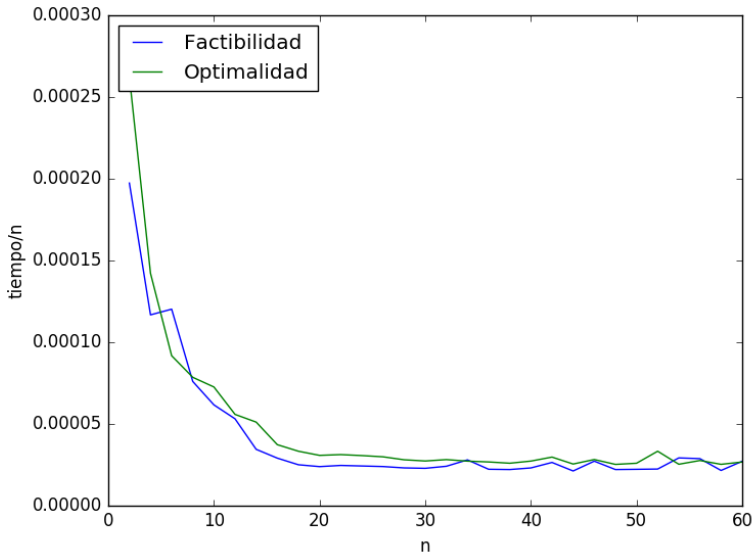


Figura 6: Mejor caso para las podas, tiempo lineal

7. Conclusión

Se llego a ver que las podas y Programación dinámica son afectados por el sort de los elementos, y el valor de ellos. También que Programación dinámica es mas afectada por el valor objetivo, que por el sort de los elementos.
Fuerza Bruta depende únicamente de la cantidad de items.
La figura 5 muestra que a medida que el valor de los elementos es mas chico, las podas empeoran y se parecen cada vez mas a Fuerza Bruta.
Para resolver este problema, Programación dinámica es el mejor de los 4 algoritmos si el valor objetivo es lo suficientemente chico para poder usarla. En cambio sí no lo es, se usaría Backtracking teniendo en cuenta el valor de los elementos y el previo ordenamiento de los mismos.

8. changelog

Se agregaron los pseudocogidos correspondientes a cada algoritmo.
Se juntaron las podas en una sección.
Se paso del algoritmo bottom-up al algoritmo top-down.
Justificación de principio de optimalidad por Absurdo.
La demostración de correctitud quedó directa despues de definir la función recursiva que calcula la solución.
Se agregó experimentación para los 4 algoritmos.