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** **3**
- 1.1. Definicion del problema Subset Sum . . . 3
- 1.2. Ejemplo de problema . . . 3
  - 1.2.1. Ejemplo 1 . . . 3
  - 1.2.2. Ejemplo 2 . . . 4
- 1.3. El objetivo del trabajo practico . . . 4

**2. Desarrollo Problema** **4**
- 2.1. Caracterización de una solución . . . 4
- 2.2. Espacio de soluciones . . . 4
- 2.3. Recorrido del espacio de soluciones . . . 5

**3. Fuerza Bruta: Recorriendo todo el espacio** **5**
- 3.1. Algoritmo . . . 5
- 3.2. Correctitud . . . 6
- 3.3. Complejidad . . . 6

**4. Backtracking: Fuerza Bruta Inteligente** **6**
- 4.1. Factibilidad: recorrido siempre válido . . . 6
  - 4.1.1. Algoritmo . . . 6
  - 4.1.2. Correctitud . . . 6
  - 4.1.3. Complejidad . . . 6
- 4.2. Optimalidad: Recorrido siempre optimo . . . 7
  - 4.2.1. Algoritmo . . . 7
  - 4.2.2. Correctitud . . . 7
  - 4.2.3. Complejidad . . . 7

**5. Programación Dinámica: Colision y Memoizacion** **7**
- 5.1. principio del Optimalidad: para Subset Sum . . . 7
- 5.2. Algoritmo . . . 8
- 5.3. Correctitud . . . 8
- 5.4. Complejidad . . . 8

**6. Experimentación** **8**
- 6.1. Hipotesis . . . 8
- 6.2. Consideraciones para las experimentaciones . . . 9
  - 6.2.1. Generación de elementos . . . 9
  - 6.2.2. Imágenes y Complejidades . . . 9
- 6.3. Complejidades Teóricas en la practica . . . 9
  - 6.3.1. Fuerza Bruta . . . 9
  - 6.3.2. Backtracking poda por factibilidad . . . 10
  - 6.3.3. Backtracking poda por optimalidad . . . 11
  - 6.3.4. Programacion dinamica . . . 11

**7. Conclusión** **12**

# 1. Introducción al Problema

## 1.1. Definicion 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.

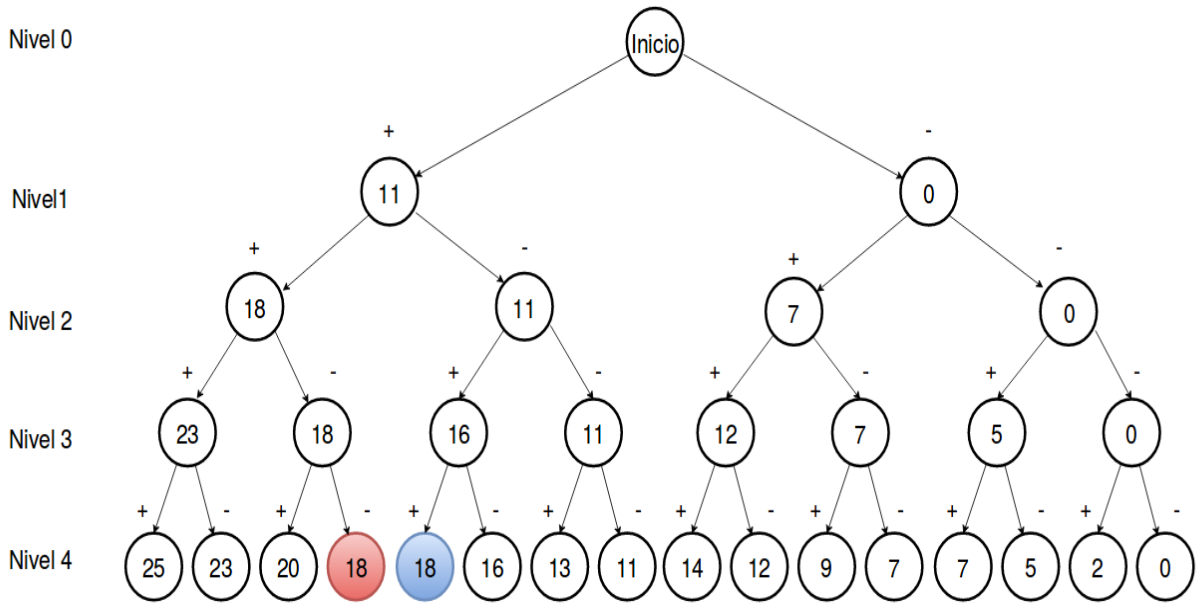
## 1.2. Ejemplo de problema

tomar el elemento(con un - arriba del nodo) para mirar si forma parte de la solución. 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 trivialmente se logra ver como cada nodo hoja es una posible solución que llego desde las decisiones tomadas).

### 1.2.1. Ejemplo 1

En este ejemplo, 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 Como se puede ver en la imagen, 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 practico

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

## 2. Desarrollo Problema

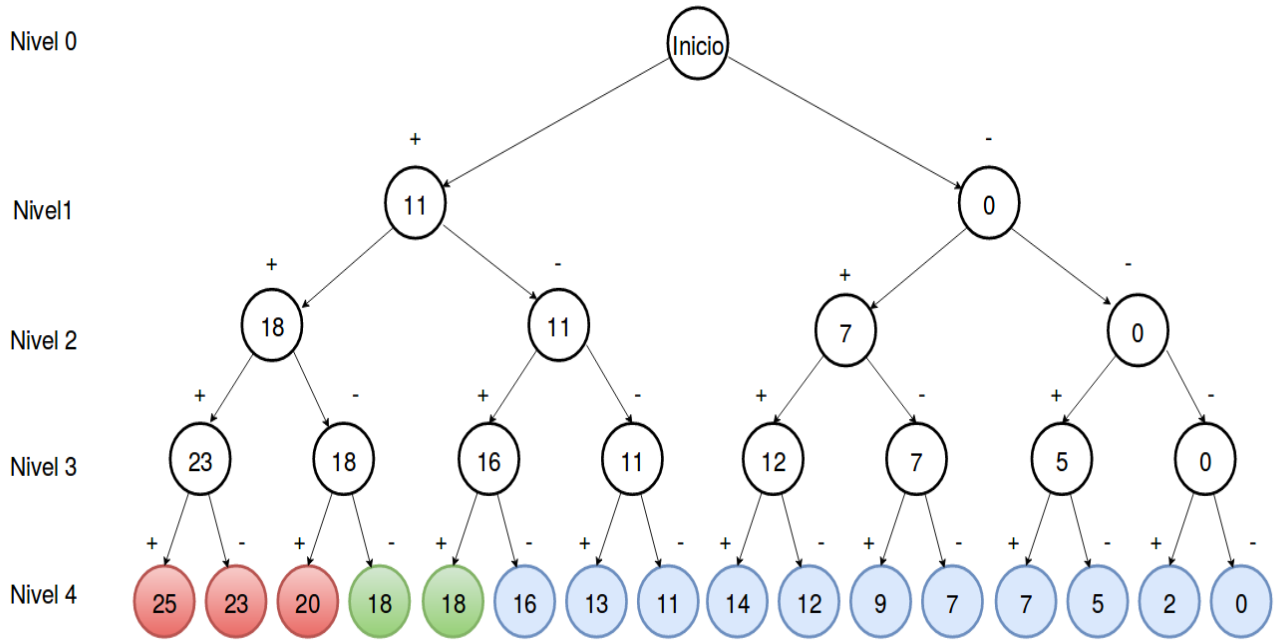
### 2.1. Caracterización de una solución

- Una posible solución es un subconjunto de elementos del conjunto original tal que la suma de ellos es menor al valor objetivo.
- Una solución factible de nuestro problema es un subconjunto de elementos del conjunto original tal que la suma de ellos es exactamente el valor objetivo.
- Una solución no factible es un subconjunto de elementos del conjunto original tal que la suma de ellos es mayor al valor objetivo

### 2.2. Espacio de soluciones

En la imagen podemos ver, que luego de diferentes decisiones, llegamos a diferentes soluciones.

- Los nodos hojas son las soluciones y están separados en tres casos.
- Los nodos celestes son las posibles 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 posibles soluciones y 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 ultima forma de pensar el problema sera ver los problemas anteriores inmediatos y ver como con ellos se puede construir el problema mas grande y gracias a memorizar los subproblemas poder obtener el siguiente de forma eficiente.

## 3. Fuerza Bruta: Recorriendo todo el espacio

### 3.1. Algoritmo

FB(vec¡int¿C, int V, int i, int n) $O(2^n)$

1 **if** $i < n$ **return** $min(1 + FB(C, V - C_i, i + 1, n), FB(C, V, i + 1, n))$ $O(1)$

2 **if** $V$ **return** 0 $O(1)$

3 **if** $i > n || V < 0$ **return** $\infty$ $O(1)$

### 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.
Es trivial que el algoritmo es correcto pues recorre todo el espacio de soluciones y solamente conserva las soluciones validas 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 seguimso 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

FB(vec¡int¿C, int V, int i, int n) O($2^n$)
1 **if** $i <= n$ O(1)
2 **if** $V == 0$ : O(1)
3 **return** 0 O(1)
4 **else** :
5 **return** $Inf^a$ O(1)
6 **return** min(FB($C$,$V$,$i+1$,$n$),1+FB($C$, $V - C_i$, $i+1$, $n$)) O($2^n$)

[a] Si el algorítmo devulve un número mayor que $n$ significa que no existe solución

#### 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: $O(2^n)$.

## 4.2. Optimalidad: Recorrido siempre optimo

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.
Dada, 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

FB(vec¡int¿C, int V, int i, int n) $O(2^n)$

```
1  if i <= n                                   O(1)
2      if V == 0 :                             O(1)
3          return 0                            O(1)
4      else :
5          return Inf[a]                       O(1)
6  return min(FB(C,V,i+1,n),1+FB(C, V − C_i, i+1, n))   O(2^n)
```

[a] Si el algorítmo devulve un número mayor que $n$ significa que no existe solución

### 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 habiá 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: $O(2^n)$.

# 5. Programación Dinámica: Colision y Memoizacion

## 5.1. principio del Optimalidad: para Subset Sum

Sea $S1$ la solucion optima usando $X_1, ..., X_{n-1}$, tal que sumen $V - X_n$
Sea $S2$ la solucion optima usando $X_1, ..., X_{n-1}$, tal que sumen $V$

Sea S = min(S1, S2), la solucion optima para el problema de n-elementos, entonces hay 2 opciones:
Usar el n-esimo elementos, o no usarlo.
Supongamos que $S$ no es solucion optima, como S no es optimo 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 optimo 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 optimo. Absurdo pues $S1$ es optimo.
- Caso $X_n$ no pertenece a $S$: $|S'| < |S| = |S2|$. Absurdo pues $S2$ es el optimo para sumar V usando $X_1, ..., X_{n-1}$

Vimos en la introducción que las soluciones de este problema pueden ser expresados mediante una secuencia de decisiones.
Ahora para que ver cumple el principio de optimalidad definimos:

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

Llamamos al problema óptimo de resolver el problema original con un elemento menos(usando el n-esimo elemento y sin usar el n-esimo elemento que son los dos únicos casos), osea dos subproblemas óptimos. Por lo tanto con esos dos subproblemas se puede armar el problema mas grande.
Con lo cual, podemos usar Programación dinámica para resolver el problema.

### 5.2. Algoritmo

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

```
1  if i <= n                                   O(1)
2      if V == 0 :                             O(1)
3          return 0                            O(1)
4      else :
5          return Inf^a                        O(1)
6  return min(FB(C,V,i+1,n),1+FB(C, V − C_i, i+1, n)) O(2^n)
```

[a] Si el algorítmo devulve un número mayor que $n$ significa que no existe solución

### 5.3. Correctitud

Dado que subset-sum cumple el principio de optimalidad, se puede usar esa forma recursiva para poder calcular el valor optimo.
Ahora veremos que el algoritmo devuelve lo mismo que la funcion.

### 5.4. Complejidad

La complejidad es fácil de deducir, se recorren dos array de cardinal V y se llama a la iteración n veces. Por lo tanto la complejidad es O(V*n)

## 6. Experimentación

### 6.1. Hipotesis

- Las podas de factibilidad y optimalidad en mejor caso tienen complejidad lineal.
- Hacer un sort de los elementos de mayor a menor, hace que las podas sean más efectivas.
- Hacer un sort de los elementos de menor a mayor, hace que las podas empeoren y sean mas parecidas a fuerza bruta

- La poda de factibilidad 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
- Fuerza Bruta depende unicamente de la cantida de items

## 6.2. Consideraciones para las experimentaciones

### 6.2.1. Generación de elementos

- Se usa la funcion sort de c++ (ordena los elementos en forma creciente)
- Los conjuntos fueron generados aleatoriamente, con la función rand de c++ (distribucion uniforme) para poder analizar en caso promedio que pasaba con cada algoritmo, en el rango entre [0..V].
- la cantidad de iteracione es igual a 50, pues se logro ver durante la experimentación que los algoritmos se estabilizaban.
- El rango para la suma fue [15000...80000], porque es limitada por Programacion Dinamica, 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 tardán demasiado.

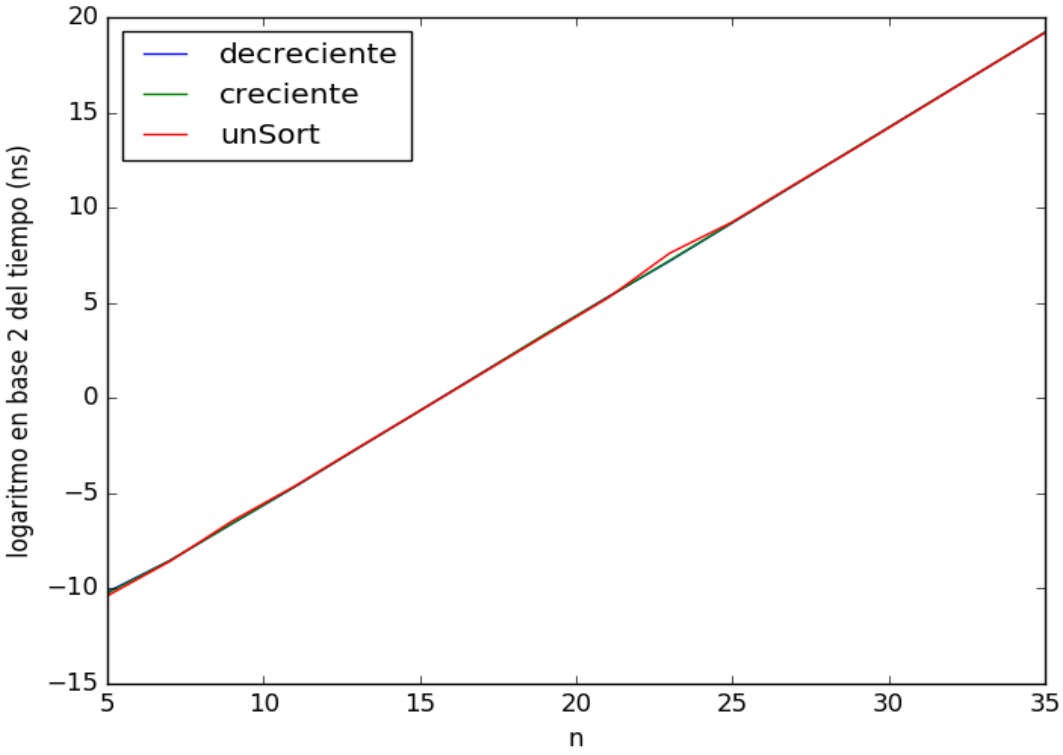
### 6.2.2. Imágenes y Complejidades

- Para medir el tiempo se tomó el promedio 50 de iteraciones
- Para todos los algoritmos se grafica el tiempo en el eje Y (en ns), y la cantidad de elementos en el eje X
- Para todos los algoritmos se dan 2 gráficos, el primero es como crece en función de la cantidad de elementos, luego el segundo algoritmo dividido por su complejidad teórica
- Todos los gráficos con excepción del último fueron hecho con la suma objetivo igual a 500(el ultimo gráfico no, para poder mostrar como afecta el valor de la suma al algoritmo de Programación dinámica)

## 6.3. Complejidades Teóricas en la practica

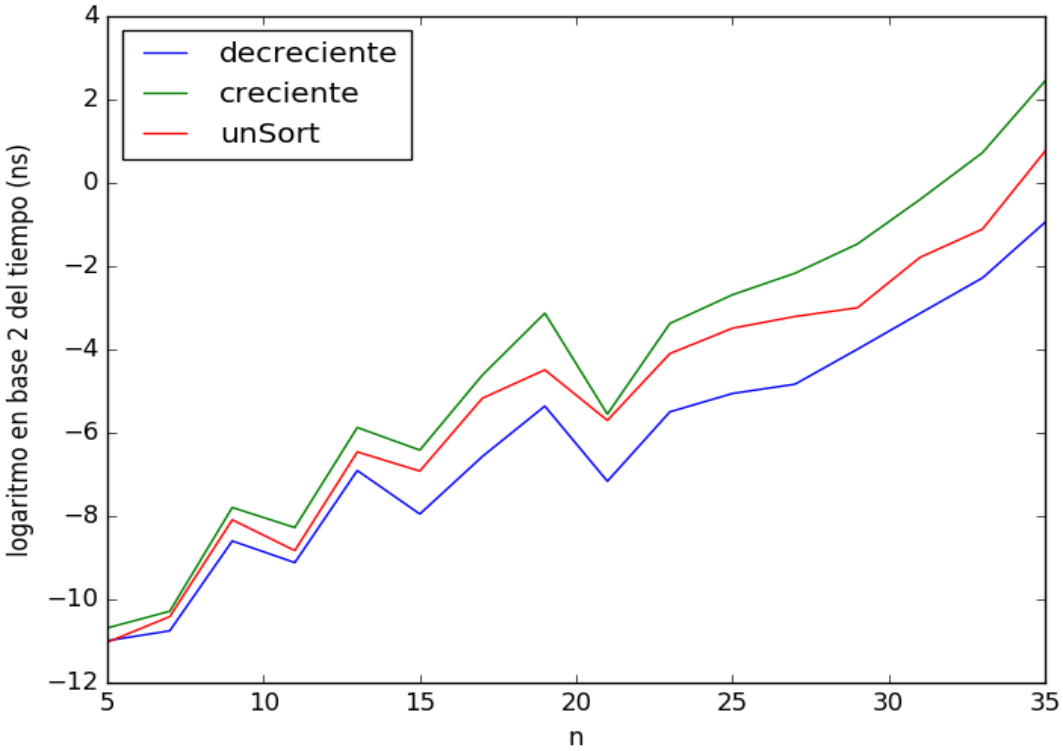
### 6.3.1. Fuerza Bruta

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.



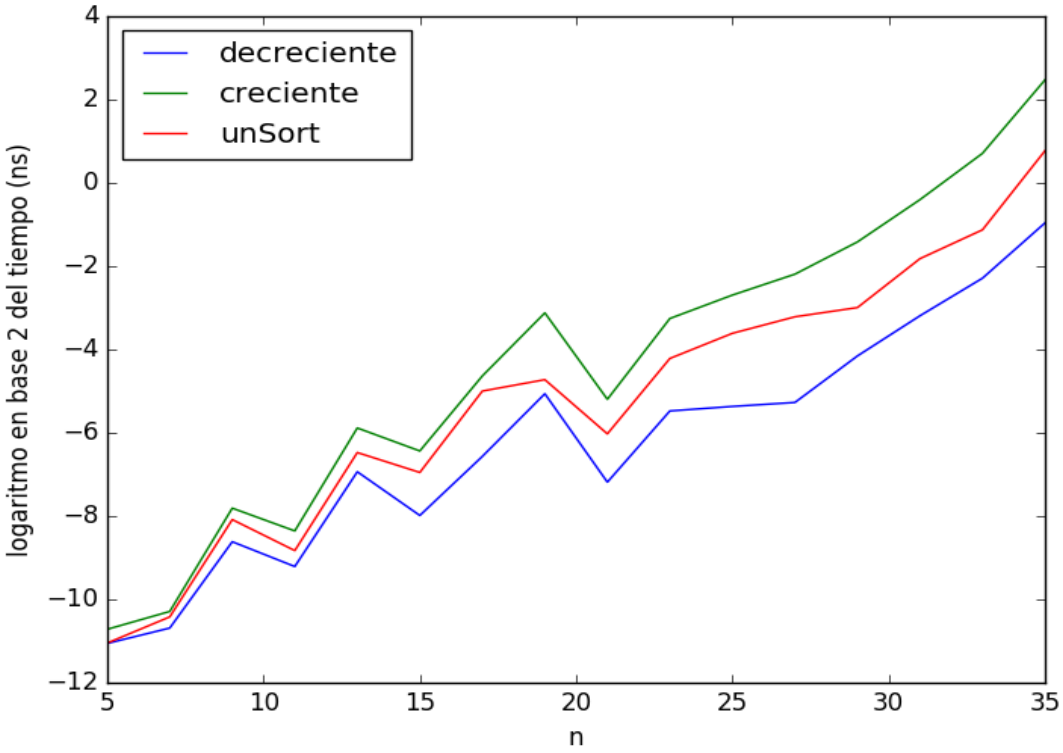

### 6.3.2. Backtracking poda por factibilidad

Tener los elementos ordenados de mayor a menor hace que la poda sea mas efectiva y empeora cuando el ordenamiento es de menor a mayor.
Se aplica logaritmo en base dos y se logra ver que el resultado es lineal.



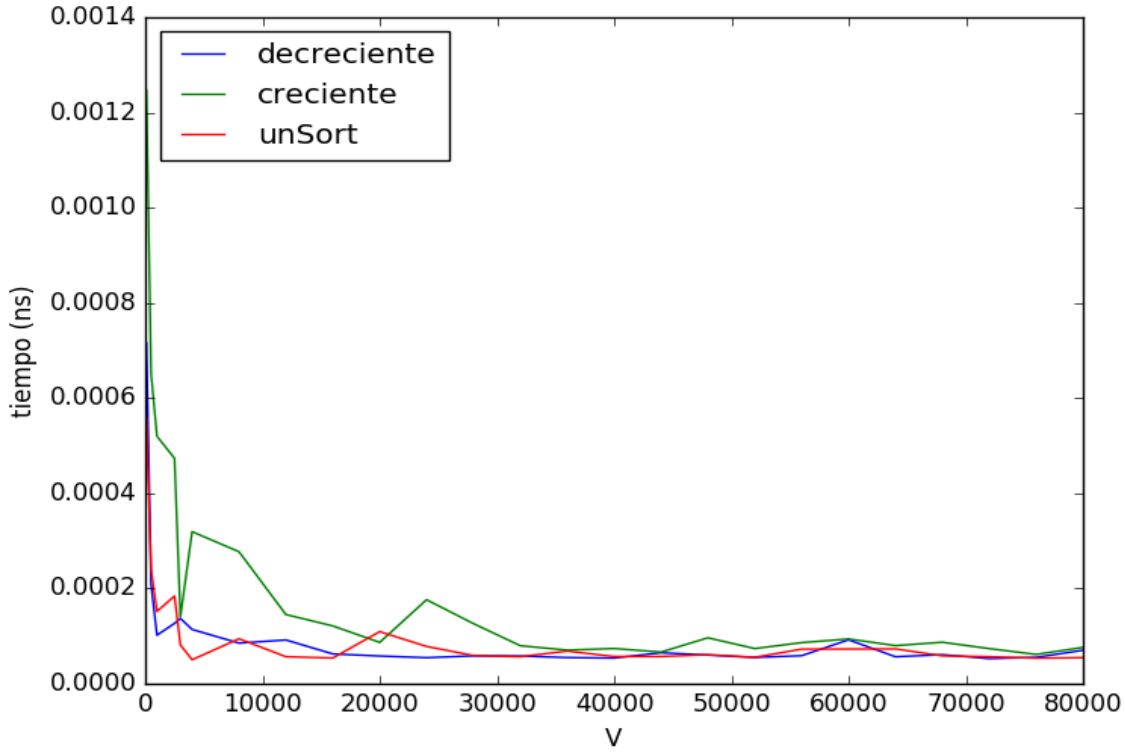
### 6.3.3. Backtracking poda por optimalidad

El ordenamiento de los elementos afecta a la poda de optimalidad y se hace el mismo analisis que para la poda de factibilidad.




### 6.3.4. Programacion dinamica

Para esta experimentacion se varia V y se compara el tiempo en funcion de el ordenamiento de los elementos.
El resultado es el esperado pues, en el grafico se ve como Programacion Dinamica es afectado directamente por el valor objetivo.




## 7. Conclusión

Los algoritmos de Backtracking no siempre siempre mejores que usar fuerza bruta.
Programación dinámica es bastante buena para valores chicos, pero logramos ver como crece rápidamente al incrementar el valor de la suma.
Como Fuerza Bruta y los algoritmos de Backtraking no son afectados por el valor de la suma, para valores muy grandes estos algoritmos podrían tener un mejor rendimiento.
Para resolver este tipo de problema, primero hay que hacer un analisis del entorno donde se va a usar, para saber si hay una cota o no, luego de eso se elige el algoritmo para ser usado. Si hay cota y es chica en comparación con la suma, se usará PD, caso contrario alguno de los algoritmos exponenciales los cuales no resultaros ser muy diferentes.