DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico I

**Eligiendo Justito**

Algoritmos y Estructuras de Datos III
Segundo Cuatrimestre 2018



| Integrante | LU | Correo electrónico |
|---|---|---|
| Imperiale, Luca | 436/15 | lucaimperiale95@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

# 1. Introduccion

## 1.1. Problema

El problema a estudiar en este trabajo práctico, va a ser el denominado *Problema de suma de subconjuntos*, o *Subset Sum*. Este consiste en dado un conjunto de numeros $C$ y un valor objetivo $V$, decidir si existe algun subconjunto de $C$, tal que todos sus elementos sumen exactamente $V$, y devolver su cardinalidad. En el caso que varios existan, la salida debera ser la cardinalidad del mas chico, y si no existen, se devolvera -1.

| Conjunto | Valor Objetivo | Resultado Esperado |
|---|---|---|
| 1 2 3 4 5 | 8 | 2 |
| 2 2 2 | 6 | 3 |
| 5 10 5 | 25 | -1 |
| 1 1 1 1 1 5 | 5 | 1 |

# 2. Desarrollo

## 2.1. Fuerza Bruta

La idea de este algoritmo es muy simple, y consiste en para cada conjunto del conjunto de partes de la entrada, o sea, para todas las posibles combinaciones de elementos del conjunto entrada, chequear una por una si suman exactamente el objetivo y devolver la cardinalidad de la mas chica.

---

**Fuerza Bruta**(**in** $\mathcal{C}$ : *conj*(`nat`),**in** *obj* : `nat`)

```
1:  min ← ∞
2:  for cada c de P(C) do                                   ▷ O(2^n)
3:      suma ← 0
4:      for cada elem en c do                               ▷ O(n)
5:          suma ← suma + elem
6:      if suma = obj then
7:          min ← minimoDe(min,|c|)
8:  if min ≠ ∞ then
9:      return min
10: else
11:     return −1
```

Complejidad: $O(|\mathcal{C}| * 2^{|\mathcal{C}|}) = O(n * 2^n)$

Justificación: Se recorren todos las conjuntos del conjunto de partes de la entrada, y para cada uno se recorren todos sus elementos. El mayor tamaño que va a poder tener alguno va a ser el mismo de la entrada, o sea $n$.

---

donde:

- $\mathcal{P}(\mathcal{C})$ es el conjunto de partes de $\mathcal{C}$
- $|\mathcal{C}|$ es el cardinal de $\mathcal{C}$
- llamamos $n$ a $|\mathcal{C}|$, osea, la cantidad de elementos de la entrada

La solucion que brinde este método va a ser correcta ya que va a recorrer todo el espacio de búsqueda del problema, asi que encontrar la solucion óptima es trivial.

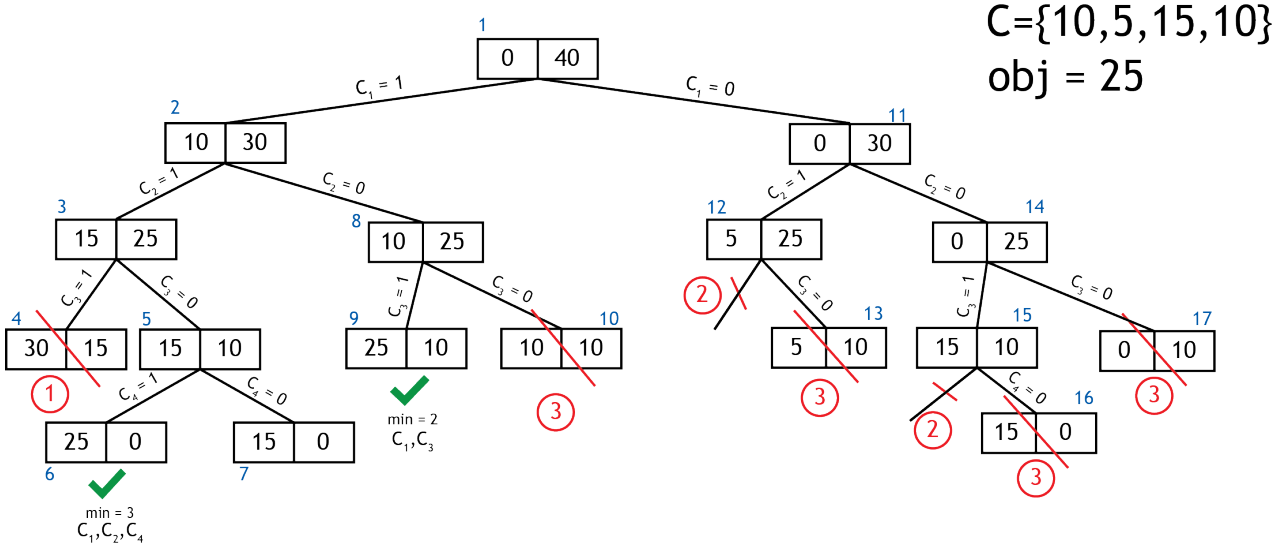
## 2.2. Backtracking

La idea de la resolucion usando la tecnica de Bactracking, es ir explorando el arbol de decisiones, tal como se hace en fuerza burta, pero pudiendo descartar ramas, o "podar",de acuerdo a diferentes criterios, y de esa forma evitar la mayor cantidad de cálculos posibles.
Las podas que utilizaremos para nuestra implementacion de backtracking son las siguientes:

- Factibilidad: Si la decision que tomo me lleva a una instancia en la cual nunca voy a poder encontrar una solucion, descarto toda esa rama.
- Optimalidad: Si la decisión que tomo me lleva a una instancia en la cual cualquier posible solucion que pueda llegar no va a ser optima en comparacion a una que ya encontré(en algún momento previo), puedo descartar toda la rama.

Para poder realizar estas podas, en el caso de nuestro problema, el algoritmo va a necesitar tener en cuenta 3 variables en todo momento: la suma parcial con los elementos que tomó esa rama, la suma de los que quedan por evaluar y el objetivo. En el ejemplo de la Figura 1 se puede ver como funciona.

Figura 1: Ejemplo de Backtracking: los numeros en azul inducan el orden de operaciones del algoritmo. En cada nodo, a la izquierda está la suma parcial hasta el momento, y a la derecha la suma de lo que queda. Los numeros en rojo indican las condiciones de corte o poda.




Las condiciones de poda en la Figura 1 son:

1. Factibilidad: cuando la sumatoria de elementos elegidos supera al valor objetivo, esa rama no va a poder tener ninguna solucion posible, por lo que se descarta.
2. Optimalidad: solo van a ser relevantes las ramas en las cuales tomo menos elementos de los que tome en la solucion minima hasta ahora, por lo que podemos descartar cualquier rama que ya tenga la misma cantidad de elementos elegidos. Cabe notar que descarto posibles soluciones con la misma cardinalidad que la ya encontrada, pero aunque las encuentre no van a importar a la solucion final.
3. Factibilidad: si entre la suma de elementos elegidos y la suma de los que quedan no llegan al valor objetivo, se puede decir que no va a haber ninguna solucion posbile en la rama, por lo que puede ser descartada.

A continuacion un pseudocódigo que muestra como podria implementarse esta idea:

**Backtracking**(**in** $\mathcal{C}$ : *conj*(`nat`),**in** *obj* : `nat`)

```
1: min ← ∞
2: total ←sumatoriaDe(C)
3: auxBacktracking(C,obj,suma=0,total)
4: if min ≠ ∞ then
5:     return min
6: else
7:     return −1
```

Complejidad:$O(n * 2^n)$

Justificación: Podemos decir que en el peor caso no va a poder realizarse ninguna poda, por lo que es facil ver que la cantidad de veces que se va a llamar a la funcion auxBacktracking va a ser $2^n$(si extendemos el arbol de decisiones de la Figura 1 vamos a tener $2^n$ nodos). Y por cada nodo se efectuan $n$ operaciones.

**auxBacktracking**(**in** $\mathcal{C}$ : *conj*(`nat`),**in** *obj* : `nat`),**in** *suma* : `nat`),**in** *resto* : `nat`)

```
1: if suma = objetivo then
2:     min ← cantidadUsados                                  ▷ O(n)
3:
4: if llegue al final del arreglo then
5:     return
6:
7: if tomando el siguiente elemento, paso las podas then
8:     auxBacktracking(siguiente elemento elegiendo el actual para la solucion)
9:
10: if sin tomar el siguiente elemento, paso las podas then
11:     auxBacktracking(siguiente elemento sin elegir el actual para la solucion)
```

Complejidad:$O(n)$

Justificación: La operacion cantidadUsados se va a fijar, para cada elemento de la entrada, si esta siendo usado en la solucion actual o no.

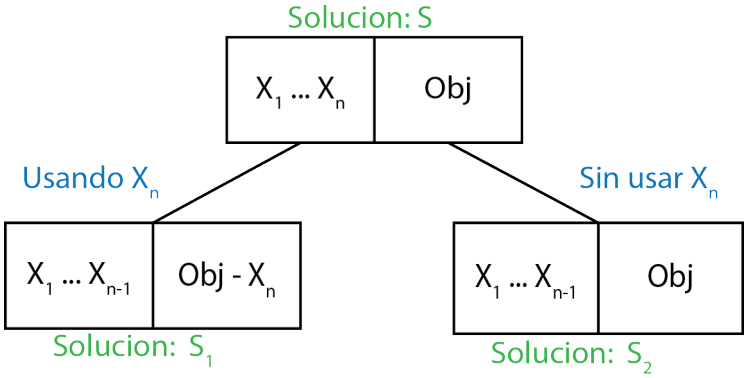
El algoritmo va a recorrer los nodos necesarios y descartando los otros, actualizando la variable *min* cuando sea necesario. Despues de todos los llamados recursivos en *min* va a quedar, de existir, la respuesta a nuestro problema. Si *min* no fue modificada, significa que la respuesta no existe, por lo que debemos devolver -1 por la especificación dada.

## 2.3. Programación Dinámica

La Programación Dinámica es una tecnica algorítmica que utiliza dos conceptos, el principio de optimalidad y el caching de soluciones ya calculadas.

El principio de optimalidad va a permitir obtener una solucion óptima al problema original a partir de soluciones óptimas de los subproblemas que se van a ir generando. Es importante que nuestro problema cumpla este principio, ya que sino no podriamos aplicar esta técnica. El siguiente esquema muestra como esto ocurre

Figura 2: Principio de Optimaldiad: cada nodo representa un problema distinto,o sea, sus elementos y el objetivo. El padre es el problema original y los hijos las dos posibilidades, usar el ultimo elemento en la solucion final o no.




En el caso que el ultimo elemento se tome en cuenta en $S$, la solucion al problema original va a ser $S = S_1 + 1$. Si el ultimo elemento no forma parte de la solucion del problema orignial, $S = S_2$.

Como estas son las unicas dos posibilidades, y nuestro algoritmo busca la menor solucion posible, $S$ sera la menor entre $S_1 + 1$ y $S_2$.

Usando esta caracterización, podemos definir una formula recursiva para resovler el problema, agregandole casos base cuando no haya mas elementos, y considerando el caso en el que el elemento es mayor al objetivo, por lo que nunca va a poder ser usado por una solución.

$$f(i, obj) = \begin{cases} 0 & i = 0, obj = 0 \\ +\infty & i = 0, obj \neq 0 \\ f\big(i-1, obj\big) & i > 0, X_i > obj \\ min\Big(f(i-1, obj), 1 + f(i-1, obj - X_i)\Big) & i > 0, X_i \leq obj \end{cases}$$

Con esta funcion ya definida, se puede ver que f(n,obj) va a ser la solucion al problema original. Esta va a hacer las llamadas recursivas necesarias hasta los casos base, y de ahi armar subsoluciones óptimas hasta conseguir la solucion final.

Puede ser que haya alguna subsolucion que deba ser calculada mas de una vez, es aquí donde entra el caching. Antes de calcular el valor de una subsolucion, primero la funcion debera chequear en memoria si ésta ya fue calculada o no. En el caso que no esté, determinará su valor y lo guardará, y si ya estaba calculado, solo usará ese valor.

A continuacion, un pseudocodigo de un algoritmo que usa Programacion Dinámica y los conceptos anteriormente descriptos.

---

**Dinamica**(**in** $\mathcal{C}$ : $conj$(`nat`),**in** $obj$ : `nat`)

1: $yaCalculados \leftarrow$ matriz de $(obj + 1)$x$(|\mathcal{C}| + 1)$ dimensiones
2: $yaCalculados \leftarrow$ lleno de "no calculados"
3: $res \leftarrow$ auxDinamica$(\mathcal{C}, yaCalculados, obj, |\mathcal{C}|)$
4: **if** $res = \infty$ **then**
5: **return** -1
6: **else**
7: **return** $res$

Complejidad:$O(n * obj)$

Justificación: En el peor caso de este algoritmo, va a ser necesario llenar la matriz de subproblemas calculados, y como cada uno tiene una complejidad de $O(1)$ cuando llega a su caso base, la complejidad del algoritmo será las dimensiones de esa matriz, o sea, $O(n * obj)$

---

**auxDinamica**(**in** $\mathcal{C}$ : *conj*(`nat`),**in** *yaCalculados* : `matriz(nat)`,**in** *obj* : `nat`,**in** *indice* : `nat`)

1: **if** (*yaCalculados*[*obj*][*indice*] no esta calculado) **then**
2: **if** (no me quedan elementos ni objetivo) **then**
3: *yaCalculados*[*obj*][*indice*] $\leftarrow 0$
4:
5: **if** (no me quedan elementos pero si objetivo) **then**
6: *yaCalculados*[*obj*][*indice*] $\leftarrow \infty$
7:
8: **if** (quedan elementos y el siguiente no puede ser usado) **then** ▷ No puede ser usado por ser mayor al objetivo
9: *yaCalculados*[*obj*][*indice*] $\leftarrow$ auxDinamica($\mathcal{C}, yaCalculados, obj, indice - 1$)
10:
11: **if** (quedan elementos y tengo que decidir si usar o no el siguiente) **then** ▷ Es menor o igual al objetivo
12: *yaCalculados*[*obj*][*indice*] $\leftarrow$ minimoDe($noUsandoElementoSiguiente, usandoElementoSiguiente$)
13: **return** *yaCalculados*[*obj*][*indice*]

donde:
$noUsandoElementoSiguiente \equiv$ auxDinamica($\mathcal{C}, yaCalculados, obj, indice - 1$)
$usandoElementoSiguiente \equiv 1 +$ auxDinamica($\mathcal{C}, yaCalculados, obj - \mathcal{C}[indice - 1], indice - 1$)

# 3. Experimentación

## 3.1. Fuerza Bruta y Backtracking

Primero para cada algoritmo analizaremos si la complejidad calculada teoricamente es realmente la que muestra en peor caso nuestra implementación.

En el caso de Fuerza Bruta, el peor caso es también el caso promedio, ya que siempre hace la misma cantidad de iteraciones para cualquier entrada del mismo tamaño. Para conseguir el peor caso de Backtracking, necesitaremos que siempre se resuelvan $2^n$ problemas, osea, que no haya podas. Esto se logra usando problemas del tipo $C_n = \{1, 1, 2^1, 2^2, 2^3, ..., 2^{n-2}\}$ con $n > 2$ y valor objetivo $= 2^{n-2}$. En este caso, el algoritmo encontrará primero la solucion sumando todos los valores hasta $2^{n-3}$, y deberá probar todas las iteraciones posibles hasta llegar a la solucion final del ultimo elemento, pues ninguna cumple ningún criterio de poda.

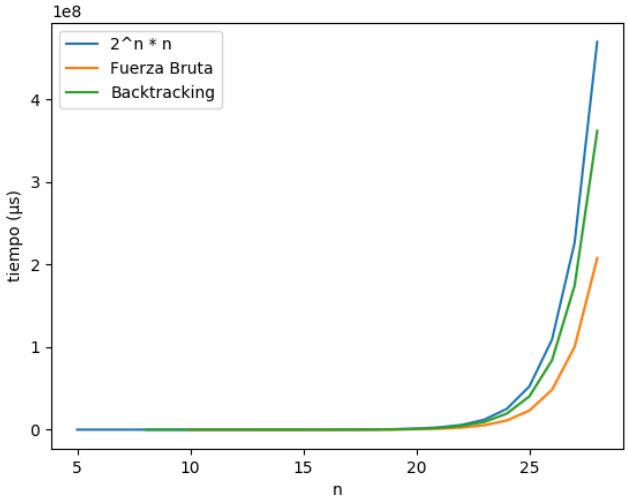



Figura 3: Comparación de los tiempos de ejecución de Fuerza Bruta y Backtracking con la complejidad calculada

En la figura 3 se puede ver que los tiempos de los dos algoritmos se comportan de manera muy similar a la funcion $f(t) = n * 2^n$, que era nuestra hipótesis.

Para no depender de la vista, podemos dividir las funciónes de tiempo por la de la complejidad que suponemos y comprobar si el resultado es una constante.

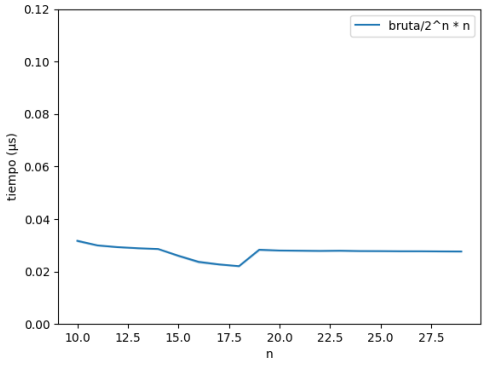



Figura 4: Cociente entre valores obtenidos de Fuerza Bruta y la funcion $n * 2^n$

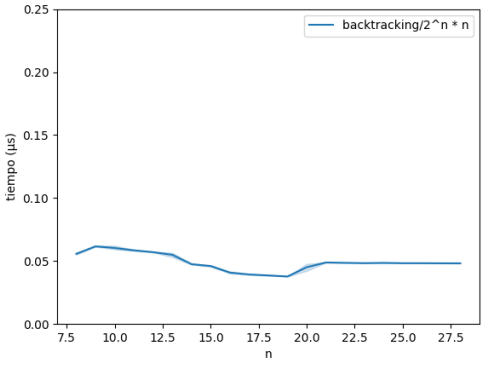



Figura 5: Cociente entre valores obtenidos de Backtracing y la funcion $n * 2^n$

Efectivamente, los cocientes se mantienen constantes en las figuras 4 y 5, lo que indica que las funciones se comportan de manera similar, o sea, que nuestras implementaciones tienen complejidad $O(n * 2^n)$.

Por último, en la Figura 6 se compara los dos algoritmos en un caso general o promedio. Caso promedio llamamos a un arreglo de n elementos aleatorios y un valor objetivo equisprobable de ser: la sumatoria de elementos de cualquier

conjunto del conjunto de partes de la entrada, asi como un valor superior a la sumatoria de todos los elementos, representando el caso en que no hay solucion. Se puede ver como el Backtracking presenta una mejora importante en comparacion de la Fuerza Bruta. Se hablará mas en detalle de cual es mejor caso para cada poda en la sección 3.2.

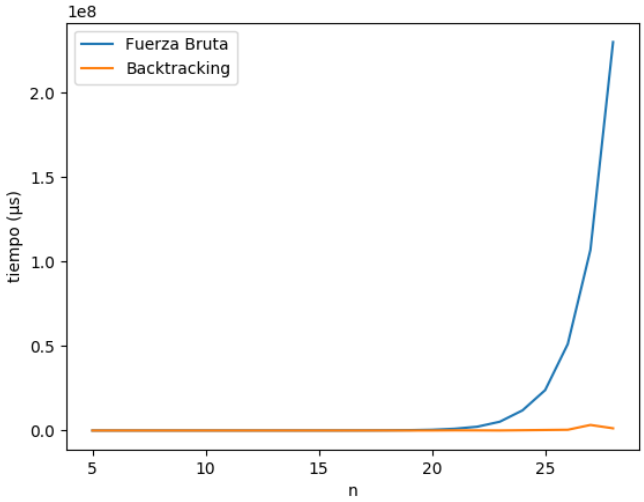



Figura 6: Comparación de los tiempos de ejecución de Fuerza Bruta y Backtracking en un caso aleatorio

## 3.2. Programación Dinámica

El peor caso de este algoritmo es cuando es necesario calcular cada subproblema posible, completando la matriz de yaCalculados, o sea, $O(n * m)$
Como conseguir un caso que ocurra esto es complicado, un equivalente sería forzar que la matriz se llene siempre. Y como sabemos que un subproblema solo va a necesitar valores de la matriz ,o en filas superiores, o en al misma fila pero columnas anteriores, podemos con un ciclo llenarla de izquiera a derecha y de arriba a abajo y que el programa siga siendo correcto, consiguiendo la misma solucion. Esta forma de completar la matriz de subproblemas es llamada Bottom Up, mientras que la que anterior es Top Down. Es importante señalar que en respecto a correctitud las dos implementaciones van a ser equivalentes, siempre vamos a llegar al mismo resultado, y una representa el peor caso de la otra.

Ya establecido el peor caso, podemos ver si el algoritmo tiene la complejidad esperada. Similar a las otras dos técnicas, vamos a dividir la funcion obtenida por la funcion de complejidad y ver si resulta una constante.

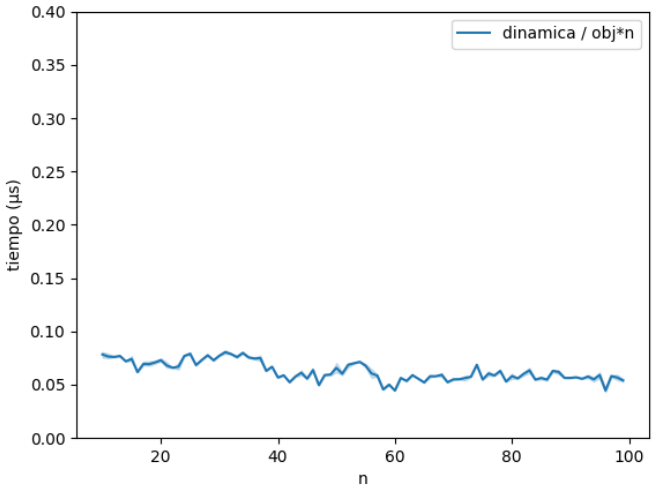



Figura 7: Cociente entre valores obtenidos y la multiplicacion del valor objetivo y la cantidad de elementos

En la figura 7 vemos como en efecto esto ocurre. Podemos tambien asegurarnos esto realizando otra prueba. Dejando fijo un alguna de las variables $n$ o $m$, y dividiendo a los valores obtenidos por ella, nos deberia quedar una funcion lineal con respecto a la variable no fija. Se puede ver esto en la las figuras 8 y 9.

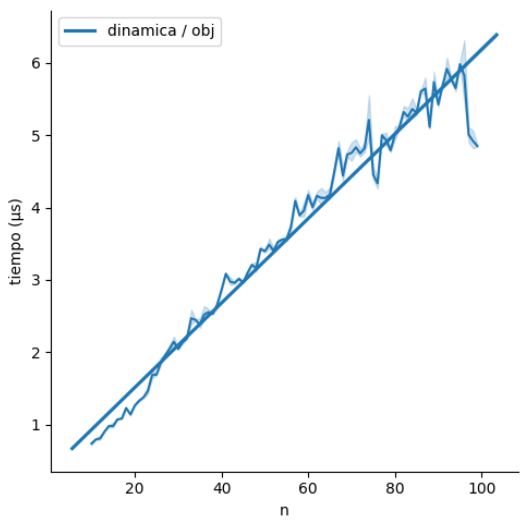



Figura 8: Cociente entre valores obtenidos y el valor objetivo, con n fijo

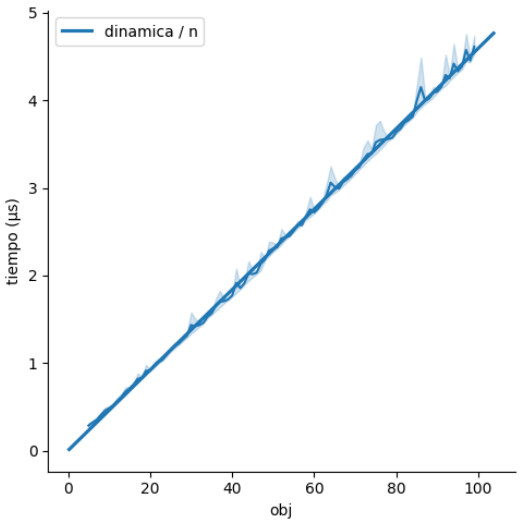



Figura 9: Cociente entre valores obtenidos y n, con el objetivo fijo

Ahora podemos asegurar que nuestra implementacion del algoritmo tiene una complejidad de $O(n * m)$.

### 3.3. Comparacion Backtracking y Dinámica

En un principio podemos hacer una comparacion general en un caso promedio.

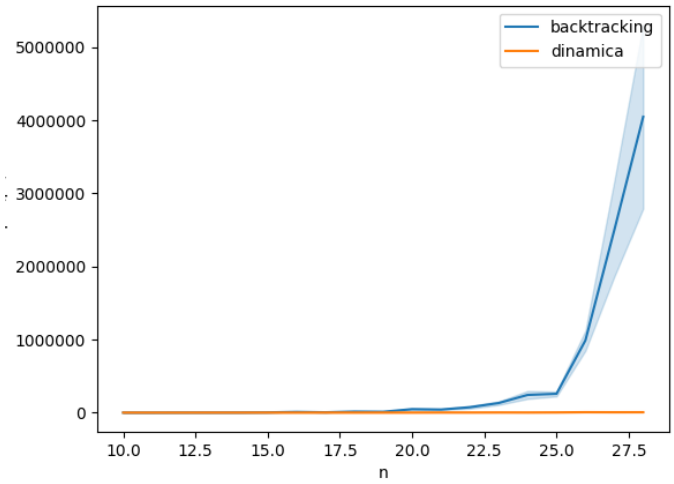



Figura 10: Tiempo de ejecucion en caso generado aleatoriamente para Programación Dinámica y Backtracking

Podemos ver que en un caso general o aleatorio, la dinámica va a tener tiempos mejores que el backtracking(Figura 10). Pero existen casos particulares en que esto no ocurre. Debido al funcinamiento de las podas en Backtracking, va a haber casos en los que se descarten muchas posibles soluciones sin siquiera evaluarlas.

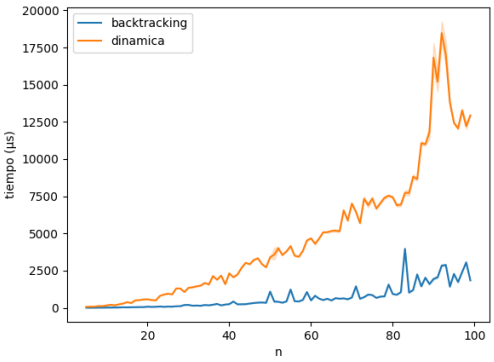



Figura 11: Tiempo de ejecucion en el mejor caso para la poda de factibilidad de Backtracking, valor objetivo igual a la suma de todos los elementos

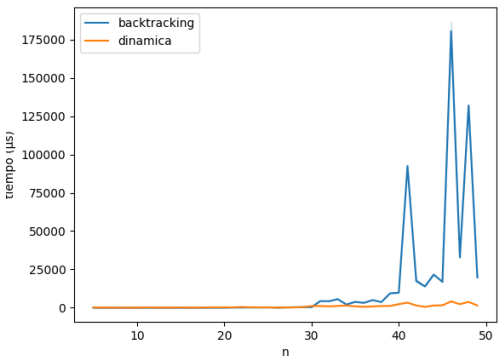



Figura 12: Tiempo de ejecucion en el mejor caso para la poda de opimalidad de Backtracking, valor objetivo igual al primer 10 % de los elementos

El caso borde de la poda de factibilidad va a ser cuando el valor objetivo sea la suma de todos los elementos del arreglo(Figura 11). Primero se va a llegar a la solucion utilizando todos los valores, y luego se van a descartar todas las soluciones usen $n - 1$ elementos porque entre los elementos usandos y los que quedan por decidir no van a sumar el valor objetivos.

Por el lado de optimalidad, el mejor caso va a ser cuando se encuentre una solucion al principio, y que tenga pocos elementos, por ejemplo, si el valor objetivo es el primer o primeros elementos de la entrada(Figura 12). En ese caso no será necesario evaluar todas las soluciones que tengan más elementos, porque no van a ser optimas en compraracion a las que ya conseguimos, así descartando mucha parte del problema. Igualmente en este caso solo conseguiremos una mejora en comparacion a la dinámica en entradas de $n$ menores a 30, lo que nos dice que el backtracking va a ser mejor cuando el objetivo es la suma de hasta los 3 primeros elementos.

# 4. Discusión y Conclusiones

Se pudo ver y comprobar empiricamente la complejidad de los tres algoritmos propuestos, asi como ver las diferencias de cada uno, en cuanto a la complejidad de codigo de su implementacion, o la complejidad temporal en sus mejores o peores casos.

Como conclusión podemos decir que el algoritmo de Programación Dinámica tiene mejor complejidad temporal que los otros, salvo en algunos casos puntuales y bordes, pero demasiado específicos como para ser relevantes. Igualmente se vio como las podas del Backtracking son una buena herramienta para mejorar el caso de Fuerza Bruta, aunque dependan en cierta parte de los datos de entrada.