Análise de Algoritmos 1º / 2020
Izabela Ramos Ferreira - 2012130024
github: https://github.com/MissHead/analise_algoritmos/tree/master/algoritmos_subsetsum

Subset Sum

Problema : Dado um conjunto de n números a[i] que a soma total é igual a M, e para algum K≤M, se existe um subconjunto dos números tais que a soma desse subconjunto dá exatamente K?

A seguir serão apresentadas 3 soluções: Recursiva, Backtracking e Iterativa.

Recursivo (código):

https://github.com/MissHead/analise_algoritmos/blob/master/algoritmos_subsetsum/subsetsum_recursivo.py

Resultados:

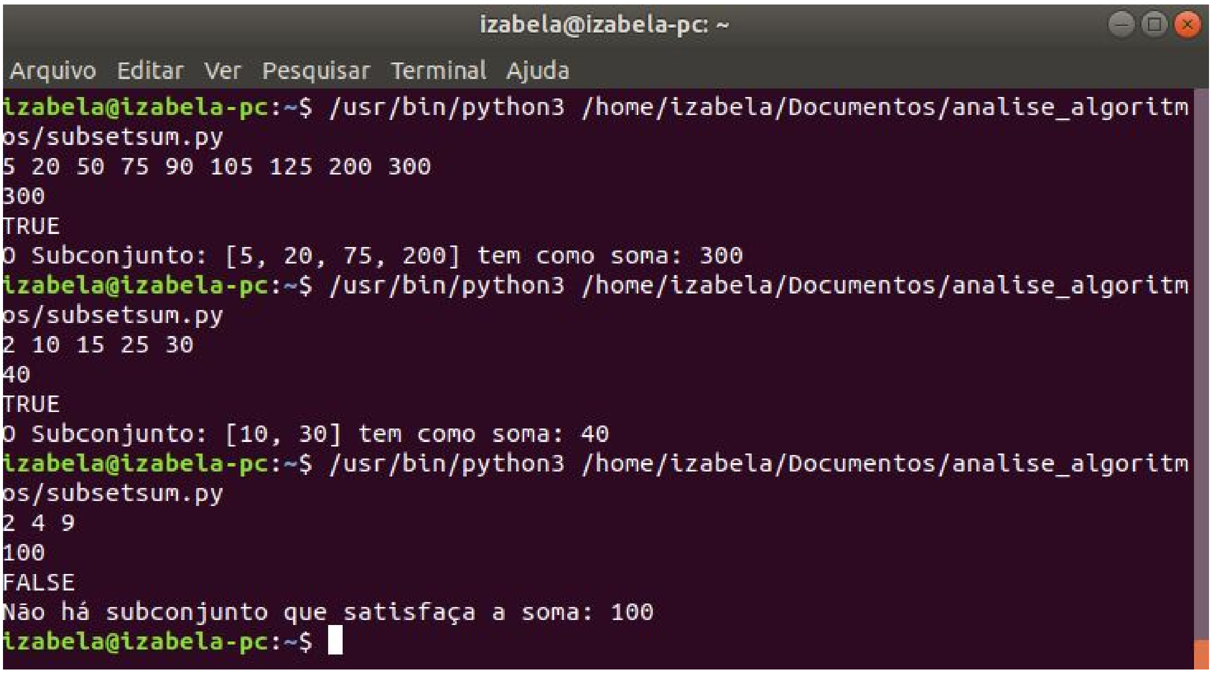


Figura 1 - Resultados da execução do algoritmo recursivo da soma dos subconjuntos

Recursivo (Análise de Complexidade):

$T(n) = 2 + T(n-1) + T(n-1)$

$T(n-1)$ é $2 + 2T(n-2)$
$T(n-2)$ é $2 + 2T(n-3)$

Portanto $T(n) = 2 + 2T(n-1)$:

$T(n) = 2 + 2(2 + 2T(n-2)) = 2 + 2^2 + 2^2T(n-2)$
$T(n) = 2 + 2^2 + 2^2(2 + 2T(n-3)) = 2 + 2^2 + 2^3 + 2^3T(n-3)$

Encontrando padrão:

$T(n) = 2 + 2^2 + ... + 2^{(k-2)} + 2^{(k-1)} + (2^k) + (2^n) * T(n-k)$

Reorganizando e reescrevendo a sequência numérica iniciada em 2:

$T(n) = (2^n) * T(n-k) + 2 + 2^2 + ... + 2^{(k-2)} + 2^{(k-1)} + (2^k)$
$T(n) = (2^n) * T(n-k) + (2^k) - 2$

Visto que a sequência:

$S = 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + ... + 2^n$
$2S = 2^2 + 2^3 + 2^4 + 2^5 + 2^6 + ... + 2^{(n+1)}$

Logo 2S - S:

$$2^2 + 2^3 + 2^4 + 2^5 + 2^6 + \ldots + 2^{(n+1)}$$

-

$$2^1 + 2^2 + 2^3 + 2^4 + 2^5 + \ldots + 2^n$$

-2 + 2^(n+1) ou ainda 2^(n+1) - 2

Aplicada à dimensão do cálculo do comportamento daquele algoritmo, a sequência é "simplificada" como (2^k) - 2

Assumindo n - k = 0, já que o consumo de tempo do algoritmo é proporcional ao número de invocações e em seu caso especial:

- T(0)=0 para n = 0
- o próprio T(n) para n >= 1

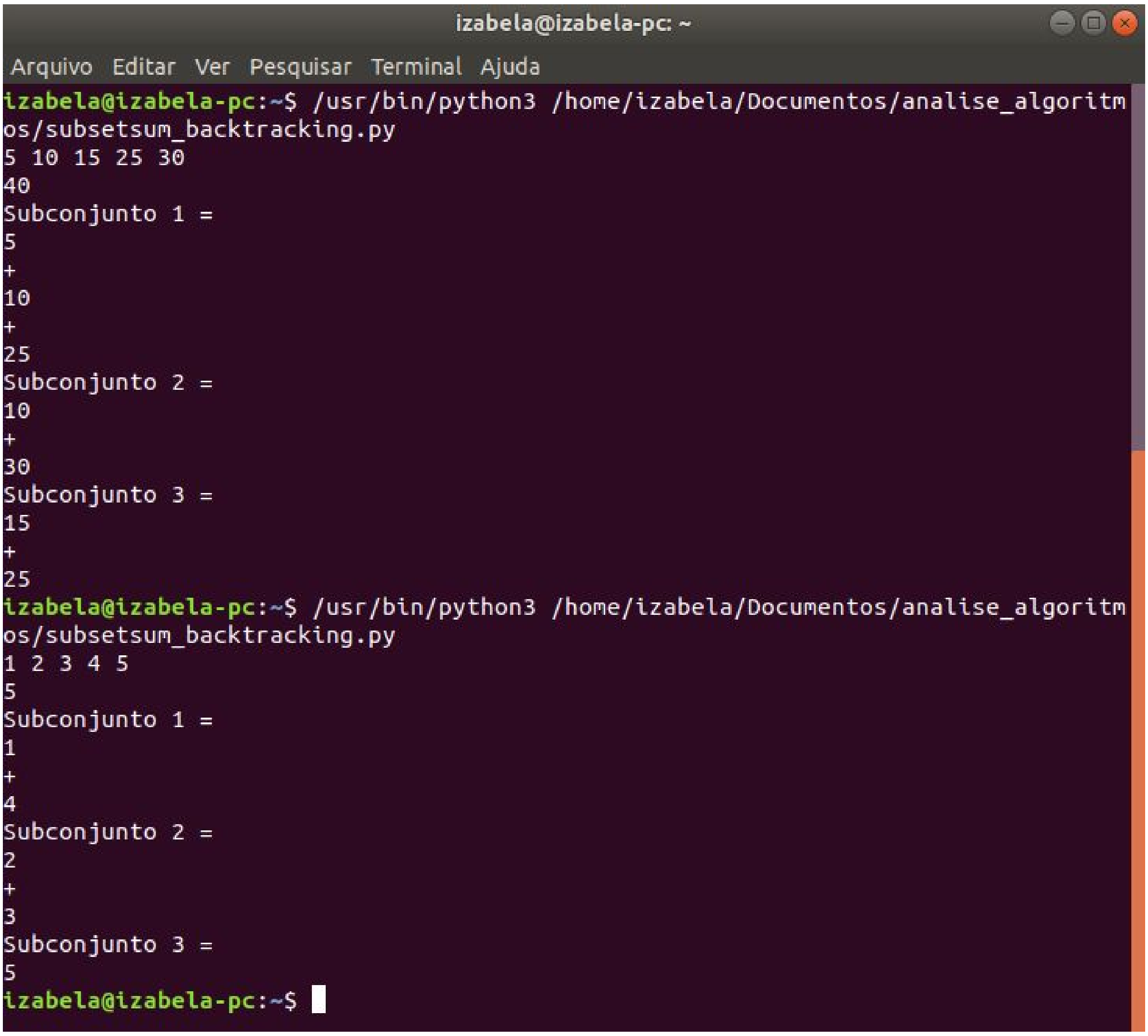
T(n) = (2^n) * constante + (2^k) - 2
T(n) = (2^n) + (2^n) - 2
T(n) = 2^(n+1) - 2

Portanto:
Consumo de tempo em até O(2^n)

Backtracking (código):

https://github.com/MissHead/analise_algoritmos/blob/master/algoritmos_subsetsum/subsetsum_backtracking.py

Resultados:



```
izabela@izabela-pc: ~
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda
izabela@izabela-pc:~$ /usr/bin/python3 /home/izabela/Documentos/analise_algoritm
os/subsetsum_backtracking.py
5 10 15 25 30
40
Subconjunto 1 =
5
+
10
+
25
Subconjunto 2 =
10
+
30
Subconjunto 3 =
15
+
25
izabela@izabela-pc:~$ /usr/bin/python3 /home/izabela/Documentos/analise_algoritm
os/subsetsum_backtracking.py
1 2 3 4 5
5
Subconjunto 1 =
1
+
4
Subconjunto 2 =
2
+
3
Subconjunto 3 =
5
izabela@izabela-pc:~$
```

Figura 2 - Resultados da execução do algoritmo com backtracking da soma dos subconjuntos

Representação da Árvore:

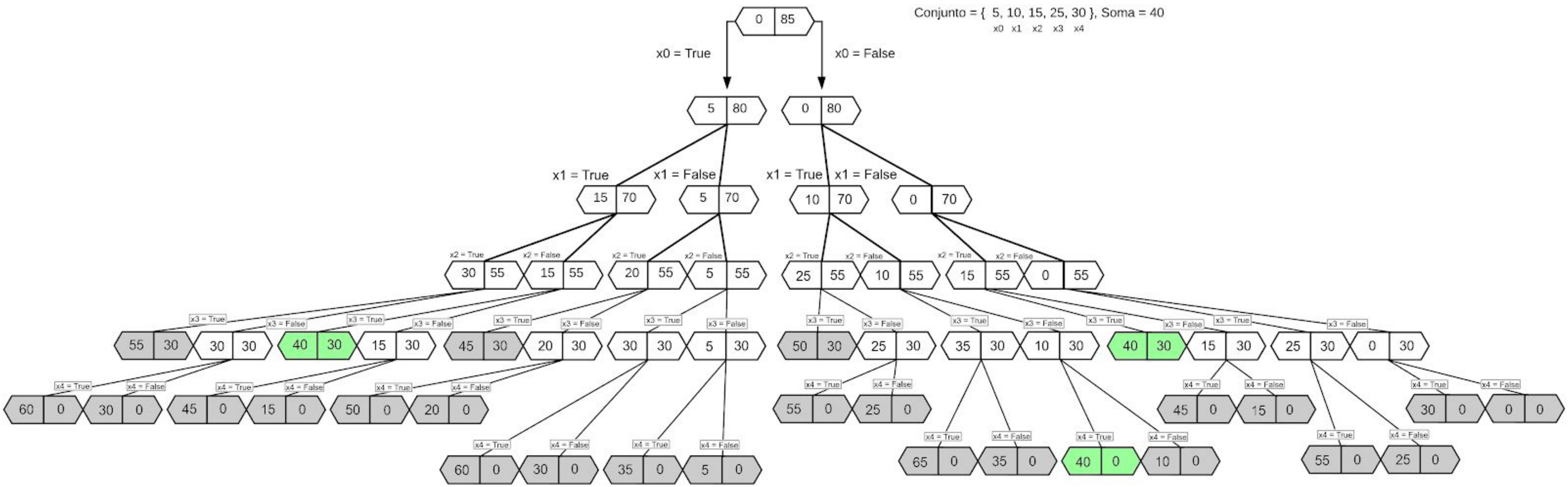


Figura 3 - Representação de árvore para o conjunto {5, 10, 15, 25, 30} e soma = 40

Backtracking (Análise de Complexidade):

Dados n números positivos, um conjunto A tal que n >= 1 esse problema exige a localização de todos os subconjuntos de A cuja soma é S.

Por exemplo, se n = 4, (A1, A2, A3, A4) = (11, 13, 24, 7) e S = 31 então os subconjuntos desejados são (11, 13, 7) e (24, 7).

Ao em vez de representar o vetor solução pelos subconjuntos de A que tenham como soma S, podemos representar o vetor de solução fornecendo os índices desses A's.

Assim, Agora as duas soluções são descritos pelos vetores (1, 2, 4) e (3, 4). Observando bem, o padrão que se toma é que todas as soluções são k-tuplas, ou seja, é uma sequência ordenada de n elementos, que pode ser definida pela recursão do par ordenado: (X1, X2, ..., Xk) levando em consideração 1 <= k <= n.

Logo, soluções diferentes podem ter tuplas de tamanhos diferentes. O problema da soma dos subconjuntos, cada solução subconjunto é representada por uma k-tupla (X1, X2, ..., Xk) tal que Xk pertença à notação {0,1} e 1 <= k <= n. Portanto Xk = 0 se o Ak não for escolhido e Xk = 1 caso Ak seja escolhido.

As soluções para a instância acima são (1, 1, 0, 1) e (0, 0, 1, 1) e essa formulação expressa todas as soluções usando uma tupla de tamanho fixo. Assim pode-se verificar que, para ambos nas formulações acima, o espaço da solução consiste em 2^n tuplas distintas.

Portanto:
Consumo de tempo em até O(2^n)

Iterativo (código):

https://github.com/MissHead/analise_algoritmos/blob/master/algoritmos_subsetsum/subsetsum_dinamico.py

Resultados:

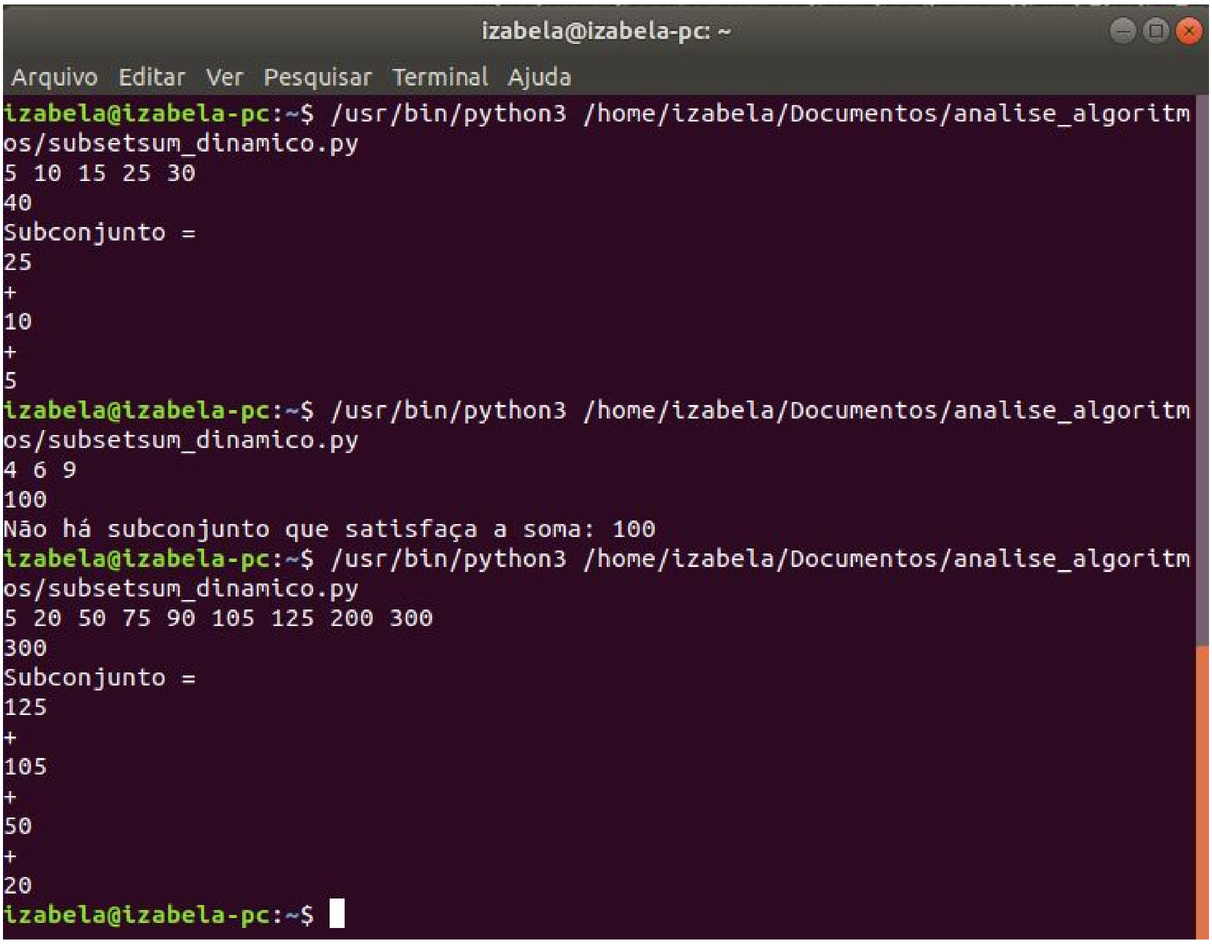


Figura 4 - Resultados da execução do algoritmo dinâmico da soma dos subconjuntos

Iterativo (Análise de Complexidade):

Programação Dinâmica

O Problema da Soma do Subconjunto é o seguinte: Dado um conjunto S de n números inteiros positivos e um valor positivo T *(target)*, determine se existe um subconjunto de S cujos elementos somam T. Para esse problema podemos considerar uma matriz M tal que:

$$M_{(i,j)} = \begin{cases} 1\,, & \text{se } i = 0 \\ 0\,, & \text{se } i < 0 \text{ ou } j = 0 \\ M(i - x_j\,,\, j-1) \text{ or } M(i\,,\, j-1)\,, & \text{caso contrario} \end{cases}$$

Assim M(T,n) resolve a declaração original do problema.

Esse algoritmo de programação dinâmica resolve cada subproblema exatamente uma vez e o espaço de todos os subproblemas pode ser representado por uma matriz i x j sendo a complexidade de tempo deste algoritmo é equivalente ao número total de elementos da matriz.

Com T linhas e n colunas, existem $T*n$ subproblemas. Portanto, a complexidade de tempo e espaço desse algoritmo de programação dinâmica é $O(T*n)$.

Para uma avaliação mais completa da complexidade do tempo, considere N o número de somas distintas que devem ser criadas com o conjunto fornecido, para que os casos a seguir possam explicar seu comportamento no tempo pseudopolinomial. Na pior das hipóteses, T é maior ou igual à soma de todos os elementos no conjunto. Como resultado, o algoritmo deve visitar todas as somas possíveis para determinar se pode ser alcançado.

Pela teoria elementar dos números, existem $(2^n) - 1$ subconjuntos não vazios distintos:

$$N <= (2^n) - 1 = O(2^n)$$

Portanto, a complexidade do tempo original $O(T*n)$ (onde $N = T$) pode ser reescrita como $O(n*(2^n))$.

Esse algoritmo de programação dinâmica é considerado pseudopolinomial porque se comporta como um algoritmo de tempo polinomial para mais elementos em S do que em T, mas na verdade não é um tempo polinomial, como demonstrado. Por isso seu tempo de execução é $O(n*(2^n))$ porque representa as piores condições, de acordo com a ordem de análise de crescimento, e não se pode garantir que T esteja de fato limitado pela soma dos elementos do conjunto.

Avaliando um caso alternativo quando T for menor que a soma de todos os elementos no conjunto, o algoritmo não considera todas as somas possíveis de subconjuntos, pois algumas seriam claramente maiores que T. Consequentemente, o algoritmo visita apenas as somas 1, 2, …, T nas piores condições para este caso em particular.

Por exemplo, considere o conjunto:

$$S = \{1, 2, 4, 8, 16\} \text{ com } T = 17;$$

$$(2^5) - 1 = 35 \text{ possibilidades de subconjuntos}$$

Porém apenas as somas 1, 2, …, 17 serão consideradas porque qualquer soma maior é desnecessária, assim $N = T = 17$. Portanto, essas condições preservam a complexidade $O(T*n)$.

Representação da Tabela:

X	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
5	V	F	F	F	F	V	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
10	V	F	F	F	F	V	F	F	F	F	V	F	F	F	F	V	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
15	V	F	F	F	F	V	F	F	F	F	V	F	F	F	F	V	F	F	F	F	V	F	F	F	F	V	F	F	F	F	V	F	F	F	F	F	F	F	F	F	F
25	V	F	F	F	F	V	F	F	F	F	V	F	F	F	F	V	F	F	F	F	V	F	F	F	F	V	F	F	F	F	V	F	F	F	F	V	F	F	F	F	V
30	V	F	F	F	F	V	F	F	F	F	V	F	F	F	F	V	F	F	F	F	V	F	F	F	F	V	F	F	F	F	V	F	F	F	F	V	F	F	F	F	V

Figura 5 - Representação da tabela para o conjunto {5, 10, 15, 25, 30} e soma = 40

Tempo de execução (Análise gráfica):

Foi executado uma sequência de testes com 17 casos de soma de subconjuntos para as 3 soluções desenvolvidas (disponíveis no github).

```
izabela@izabela-pc:~/Documentos/analise_algoritmos/algoritmos_subsetsum$
python3 subsetsum_recursivo_testes.py
Tempo Execução para SubsetSum Recursivo

1,6689300537109375e-06
1,5497207641601562e-05
4,935264587402344e-05
6,67572021484375e-06
2,5033950805664062e-05
1,9073486328125e-06
1,6689300537109375e-06
0,00023889541625976562
3,0994415283203125e-06
0,003977298736572266
0,06876683235168457
0,3854653835296631
0,0415959358215332
0,037240028381347656
0,004185676574707031
0,050749778747558594
0,0008382797241210938
```

Figura 6 - Tempo de execução dos 17 casos de teste para a solução recursiva

Figura 7 - Gráfico de desempenho da solução recursiva

```
izabela@izabela-pc:~/Documentos/analise_algoritmos/algoritmos_subsetsum$
python3 subsetsum_backtracking_testes.py
Tempo Execução para SubsetSum Backtracking

2,8371810913085938e-05
0,00010228157043457031
0,0006661415100097656
0,00012874603271484375
0,0005333423614501953
0,0002713203430175781
0,0007476806640625
0,002895832061767578
0,00690913200378418
0,00760340690612793
0,013421773910522461
0,040067434310913086
0,9342598915100098
0,10028910636901855
1,223769187927246
7,570649147033691
14,5672025680542
```

Figura 8 - Tempo de execução dos 17 casos de teste para a solução com backtracking

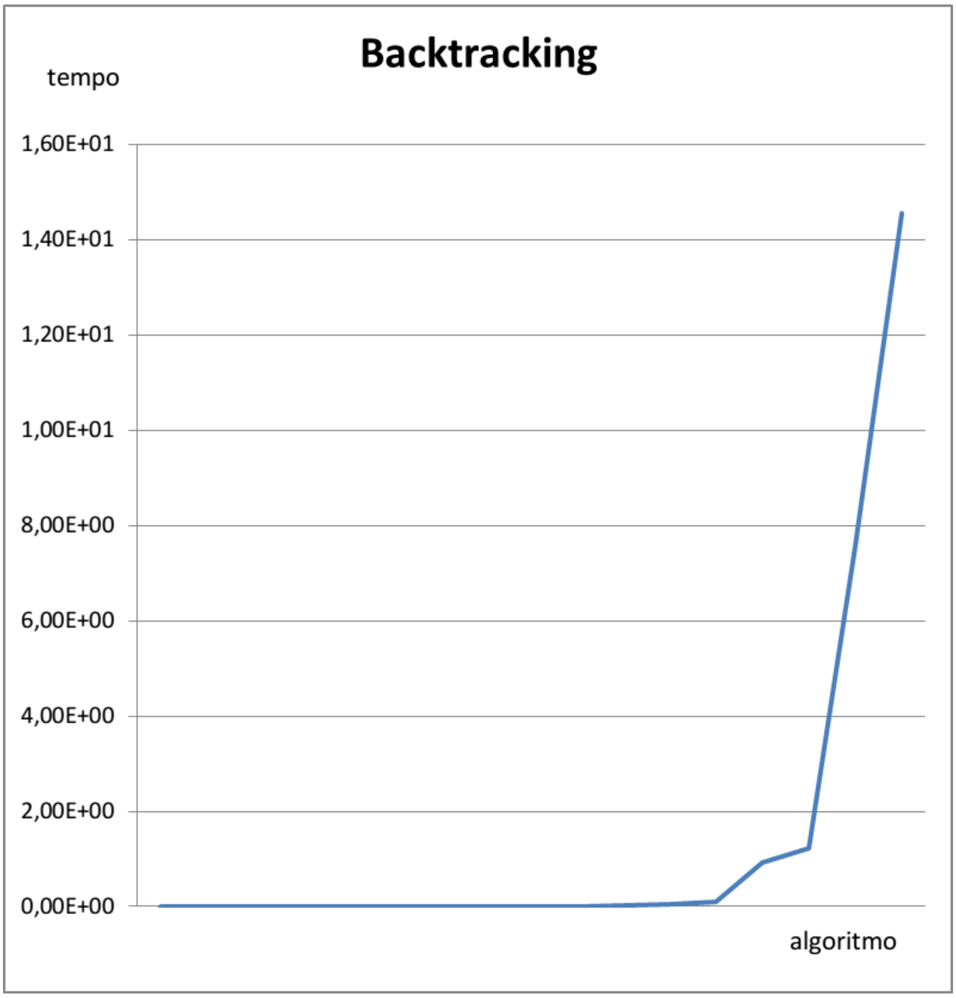


Figura 9 - Gráfico de desempenho da solução com backtracking

```
izabela@izabela-pc:~/Documentos/analise_algoritmos/algoritmos_subsetsum$
python3 subsetsum_dinamico_testes.py
Tempo Execução para SubsetSum Dinâmico

2,3365020751953125e-05
0,000148773193359375
0,0008878707885742188
0,000118255615234375
0,0003609657287597656
0,00036525726318359375
0,0004384517669677734
0,0010416507720947266
0,002008199691772461
0,0013287067413330078
0,002251148223876953
0,003512144088745117
0,051290035247802734
0,0036602020263671875
0,015650510787963867
0,059033870697021484
0,024115324020385742
```

Figura 10 - Tempo de execução dos 17 casos de teste para a solução dinâmica

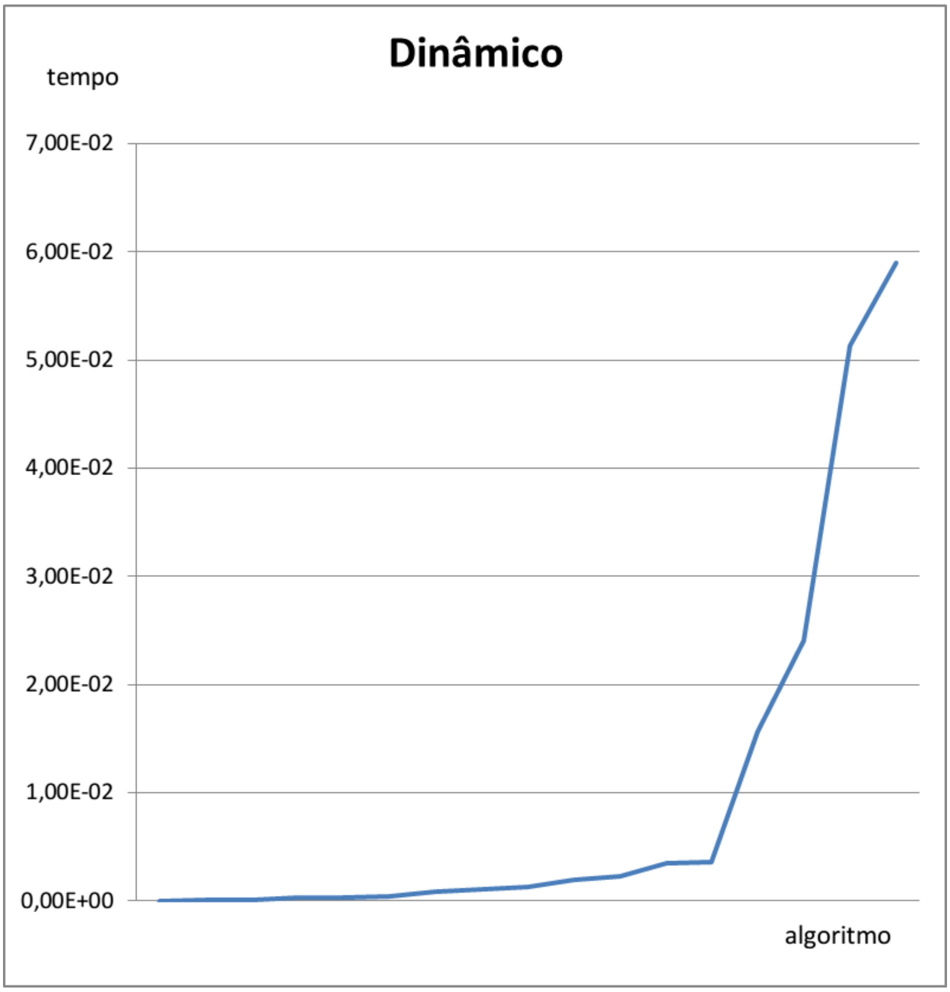


Figura 11 - Gráfico de desempenho da solução dinâmica

Quanto a cada complexidade analisada matematicamente:

Recursivo O(2^(n+1) - 2):

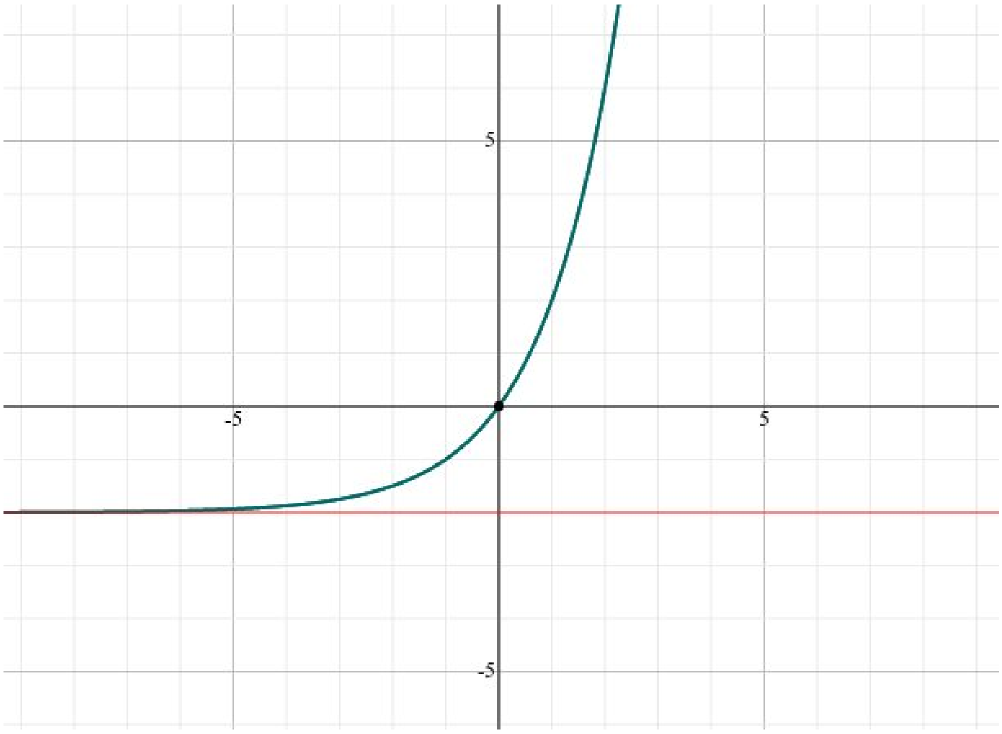


Figura 12 - Gráfico da expressão algébrica do algoritmo de soma de subconjuntos recursivo com relação ao tempo

Backtracking O(2^n):

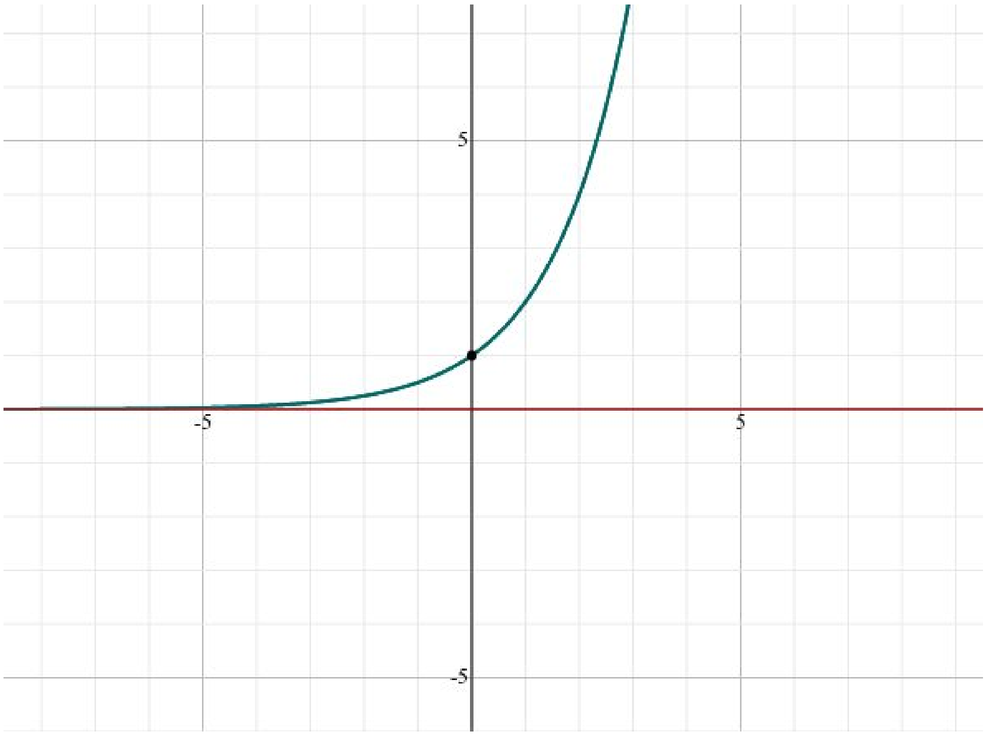


Figura 13 - Gráfico da complexidade do algoritmo de soma de subconjuntos com backtracking

Iterativo O(T*n):

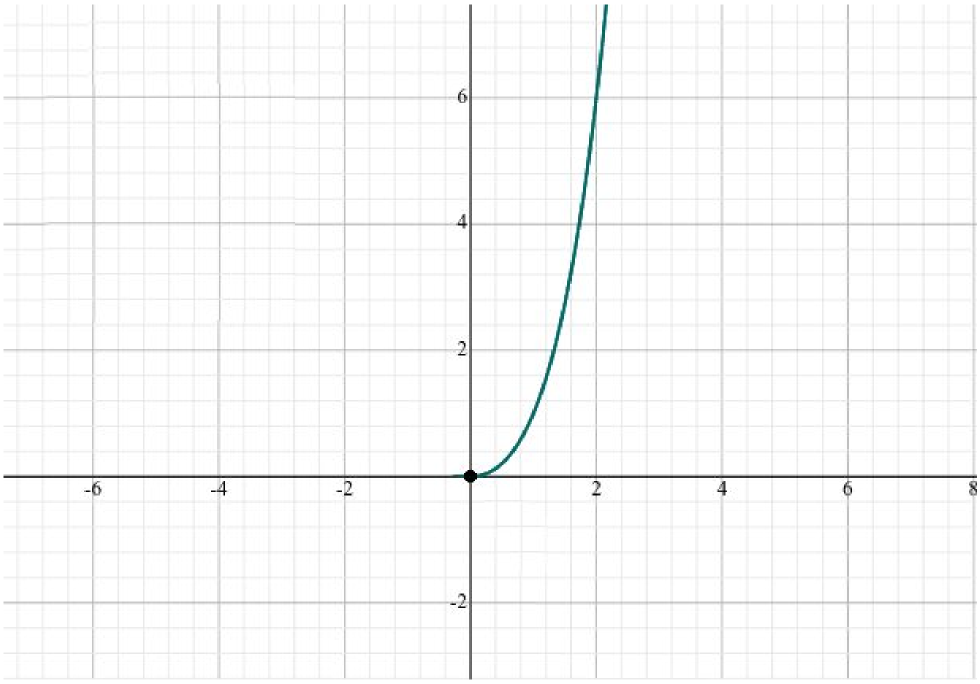


Figura 14 - Gráfico da expressão algébrica do algoritmo de soma de subconjuntos dinâmica com relação ao tempo

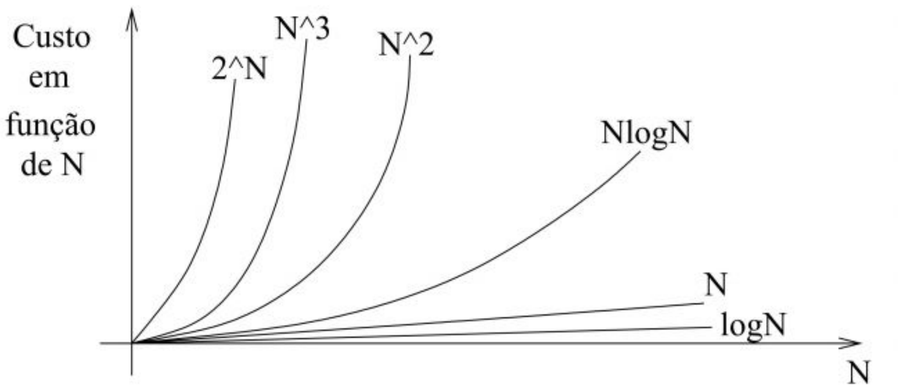


Figura 15 - Análise geral de complexidades de algoritmos

Conclusão:

É possível visualizar o comportamento do algoritmo na demonstração gráfica nas figuras 7, 9 e 11 e 12, 13 e 14 respectivamente. Observa-se a diferença de performance para cada solução e como elas correspondem às interpretações gráficas e algébricas aqui demonstrado.

Portanto, com relação a performance, a menos que os valores dos números inteiros no conjunto sejam extremamente grandes, como visto nos testes efetuados (disponíveis no github), podemos visualizar que a solução pseudo-polinomial baseada em programação dinâmica forneceu a solução otimizada mais rápida e garantida para o problema de soma de subconjuntos.