**Curso: Ciência da Computação / Engenharia da Computação**

**Disciplina: Análise de Algoritmo**

**Professor: Felippe Giuliani**

***Prova—P1***

|  |  |
| --- | --- |
| Data da atividade | Prazo para envio |
| 14/04/2020 | 21/04/2020 |

**Alunos:**

Gabriel Antônio de Vasconcellos Barreto – 1622082022

Gabriel Martins Caldas e Almeida – 1612082068

**Prova.P1 – PARTE 1**

**SubSetSum Recursivo**

a)Código em Python

def subsetsum(conjunto, capacidade, parcial=[]):  
 soma = sum(parcial)  
  
  
 if soma == capacidade:  
 encontrou = True  
 print(encontrou)  
 return True  
  
 if soma >= capacidade:  
 return  
  
 for i in range(len(conjunto)):  
 k = conjunto[i]  
 remaining = conjunto[i+1:]  
 found = subsetsum(remaining, capacidade, parcial + [k])  
 if found:  
 return True  
  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 A = [5, 4, 10, 8, 9, 6]  
 capacidade = 17  
 if subsetsum(A,capacidade):  
 print()  
 else:  
 print(False)

b)

T(n) = T(n-1) + T(n-2) + T(n-3) + ... n vezes

T(n-1) = T(n-2) + T(n-3) + T(n-4) + ...

Substituindo T(n-1) em T(n)

T(n) = T(n-1) + T(n-1) + ... n substituições

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

Substituindo n vezes

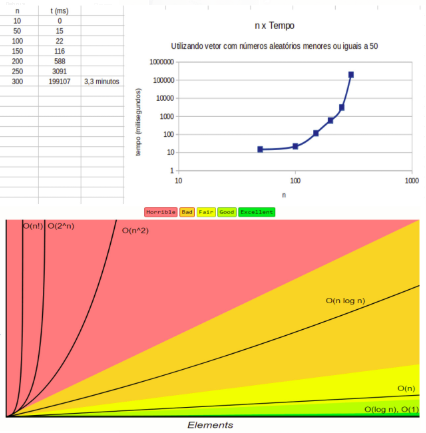
T(n) = ...2\*2\*2\*2\*2\*T(n-k)

T(n) = 2^n \* T(n-k)

Assistoticamente temos:

T(n) = O(2^n)

c) Verificando o gráfico das complexidades padrão podemos observar que o SubSetSum recursivo apresenta uma das piores complexidades das listadas(2^n), consumindo uma grande quantidade de tempo de maneira exponencial conforme o n cresce.



**Prova.P1 – PARTE 2**

**SubSetSum Dinâmico**

a)Código em Java

package analisealgp1;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

public class DynamicSubSetSum {

static boolean Subsetsum(int conjunto[], int tam, int capacidade) {

ArrayList<Integer> b = new ArrayList<>();

boolean subset[][] = new boolean[tam][capacidade + 1 ];

if(conjunto[0] > capacidade) {

return false;

}

for (int i = 0; i < tam; i++)

subset[i][0] = true;

for (int i = 1; i <= capacidade; i++) {

subset[0][i] = false;

}

for (int i = 1; i <= capacidade; i++) {

for (int j = 0; j < tam; j++) {

if(conjunto[j] == i ) {

subset[0][i] = true;

}

break;

}

}

for (int i = 1; i < tam; i++)

{

for (int j = 1; j <= capacidade; j++)

{

if( j < conjunto[i]) {

subset[i][j] = subset[i-1][j];

}else {

if(subset[i-1][j] == false) {

subset[i][j] = subset[i-1][j - conjunto[i]];

}else {

subset[i][j] = true;

}

}

}

}

int sump = 0;

int i = tam-1;

int j = capacidade;

if (subset[tam-1][capacidade]) {

while (sump != capacidade && i >=0 && j>= 0) {

if ( subset[i][j] == true && subset[i-1][j] == false ) {

b.add(conjunto[i]);

sump = sump + conjunto[i];

j = j - conjunto[i];

i = i - 1;

}

else {

i = i - 1;

}

}

}

System.out.println(b);

return subset[tam-1][capacidade];

}

public static void main(String args[]) {

int conjunto[] = {0,2,5,7,8,12}; //Para cada caso de teste incluir o '0' no ínicio do vetor

int capacidade = 17;

int tam = conjunto.length;

if (Subsetsum(conjunto, tam, capacidade) == true)

System.out.println("SubSet Encontrado");

else

System.out.println("Nenhum subset encontrado");

}

}

b) Contando instruções:

T(n) = O(n) + O(c) + O(n\*c) + O(n\*c) + O(n)

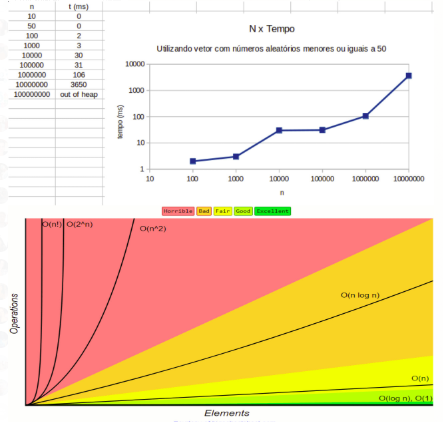
T(n) = 2O(n) + O(c) + 2O(n\*c)

T(n) = O(n) + O(c) + O(n\*c)

T(n) = O(n\*c) se n > c ou n < c

T(n) = O(n^2) se n = c

c) O SubSetSum dinâmico é a melhor alternativa das outras três, pois sua complexidade é O(n\*c) ou O(n^2), e observando o gráfico das complexidades padrão ela vai ser sempre mais eficiente que a complexidade 2^n independente de n.



**Prova.P1 – PARTE 3**

**SubSetSum BackTracking**

a)Código em Java

**package** analisealgp1\_3;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

**int** conjunto[] = {3, 34, 4, 12, 5, 2};

**int** capacidade = 17;

//int set[] = {10, 7, 5, 18, 12, 20, 15};

SubsetSumBackTracking subBack = **new** SubsetSumBackTracking(conjunto, capacidade);

subBack.subSetSumBackTracking(0,0);

**if**(subBack.existeSubConjunto == **false**) {

System.***out***.print("Não existe subconjunto que corresponda à capacidade");

}

}

}

**package** analisealgp1\_3;

**import** java.util.Stack;

**public** **class** SubsetSumBackTracking {

**int** conjunto[];

**int** capacidade;

Stack<Integer> pilhaDeSolucao;

**boolean** existeSubConjunto;

SubsetSumBackTracking(**int** conjunto[], **int** sum){

**this**.conjunto = conjunto;

**this**.capacidade = sum;

**this**.pilhaDeSolucao = **new** Stack<>();

existeSubConjunto = **false**;

}

**public** **boolean** subSetSumBackTracking(**int** somaTemp, **int** indicePilhaDeSolucao){

//Retorna false se a soma temporaria é maior que a capacidade

**if**(somaTemp>capacidade) {

**return** **false**;

}

//verifica se a pilha tem os numeros corretos para fechar a capacidade

**if**(somaTemp==capacidade){

existeSubConjunto = **true**;

mostraSubSets();

//retorna false para podermos continuar procurando

**return** **false**;

}

**for**(**int** i=indicePilhaDeSolucao; i<conjunto.length; i++){

pilhaDeSolucao.push(conjunto[i]);

//soma o elemento conjunto[i] com a soma temporaria e começa recursivamente do proximo numero

**if**(subSetSumBackTracking(somaTemp+conjunto[i],i+1)==**true**){

**return** **true**;

}

//remove elemento do topo da pilha, isto é, remove a folha da arvore

pilhaDeSolucao.pop();

}

//nenhuma combinaçao corresponde à capacidade

**return** **false**;

}

**private** **void** mostraSubSets(){

System.***out***.print("Subset: ");

**for** (**int** i =0;i< pilhaDeSolucao.size();i++){

System.***out***.print(pilhaDeSolucao.get(i)+" ");

}

System.***out***.println();

}

}

b)

T(n) = T(n-1) + T(n-2) + T(n-3) + ... n vezes

T(n-1) = T(n-2) + T(n-3) + T(n-4) + ...

Substituindo T(n-1) em T(n)

T(n) = T(n-1) + T(n-1) + ... n substituições

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

Substituindo n vezes

T(n) = ...2\*2\*2\*2\*2\*T(n-k)

T(n) = 2^n \* T(n-k)

Assintoticamente temos:

T(n) = O(2^n)

c) A comparação entre os gráficos experimentais e teóricos mostram que o algoritmo se assemelha

muito com a complexidade padrão O(2^n), ficando também atrás do SubSetSum dinâmico em questão de eficiência.

