Cálculo complexidade Subset sum Dinâmico :

Imagem do código:

```java
public void gerar_matriz_de_valores_possiveis() {
    for(int i=0;i<this.getMatriz().length;i++) {
        for(int j=0;j<this.getMatriz()[0].length;j++) {
            if(i==0) {
                if(j==0 || j== this.getVetfloat()[0]) {
                    this.getMatriz()[i][j]=true;
                }
                else {
                    this.getMatriz()[i][j]=false;
                }
            }
            else {
                if(j<this.getVetfloat()[i]) {
                    this.getMatriz()[i][j]=this.getMatriz()[i-1][j];
                }
                else if(this.getMatriz()[i-1][j]) {
                    this.getMatriz()[i][j]=true;
                }
                else if(this.getMatriz()[i-1][j-(int)this.getVetfloat()[i]]) {
                    this.getMatriz()[i][j]=true;
                }
                else {
                    this.getMatriz()[i][j]=false;
                }
            }
        }
    }
}
```

Contando as instruções:

```java
public void gerar_matriz_de_valores_possiveis() {
    for(int i=0;i<this.getMatriz().length;i++) {
        for(int j=0;j<this.getMatriz()[0].length;j++) {
            if(i==0) { -> 1
                if(j==0 || j== this.getVetfloat()[0]) { -> 2
                    this.getMatriz()[i][j]=true; ->1
                }
                else {
                    this.getMatriz()[i][j]=false; ->1
                }
            }
            else {
                if(j<this.getVetfloat()[i]) { ->1
                    this.getMatriz()[i][j]=this.getMatriz()[i-1][j]; ->1
                }
                else if(this.getMatriz()[i-1][j]) { ->1
                    this.getMatriz()[i][j]=true; ->1
                }                                                    ->1
                else if(this.getMatriz()[i-1][j-(int)this.getVetfloat()[i]]) {
                    this.getMatriz()[i][j]=true; ->1
                }
                else {
                    this.getMatriz()[i][j]=false; ->1
                }
            }
        }
    }
}
```

Considerando a matriz como y linhas e x colunas

O F(n)= 2+y(tudo que está dentro do for+2)

F(n)= 2+y(((tudo que está dentro do for+2)x+2)+2)

F(n)= 2+y((12+2)x+2+2)

F(n)= 14xy+4y+2

Considerando o n=x=y para que se tenha a complexidade do algoritmo, ou seja a matriz vai ser de nxn.

$F(n) = 14n^2 + 4n + 2$

Assim tendo complexidade O(n^2).

Cálculo complexidade Subset sum Recursivo :

Imagem do código:

```java
public boolean existe_o_somatorio_recursivo(float somatorio,int interação,String valores) {
    if(somatorio == 0) {
        add_vet_valores(valores);
        return true;
    }
    else if(interação == this.getVetfloat().length) {
        return false;
    }
    else {
        boolean ignora = existe_o_somatorio_recursivo(somatorio,interação+1,valores);
        boolean considera = existe_o_somatorio_recursivo(somatorio-this.getVetfloat()[interação],interação+1,valores+this.getVetfloat()[interação]+" ");
        return ignora || considera;
    }
}
```

Quando a função recursiva chama ela mesma duas vezes tem:

Para quando a função recursiva tiver 2 instruções:

O valor da função F(n) será:

F(n)= 2+F(n-1)+F(n-1)

F(n)=2+2F(n-1)

O valor de F(n-1) = 2+2F(n-2)

Assim substituindo no valor de F(n)

F(n)=2+2(2+2F(n-2))

$\mathrm{F(n)} = 2 + 2^2 + 2^2 F(n-2)$

Continuando substituindo o valor de F(n-2) e depois de F(n-3) e assim por diante, sempre obterá a função de F(n) como sendo $F(n) = 2 + 2^2 + 2^3 + \cdots + 2^{x-1} + 2^x + 2^x F(n-x)$

Considerando que o somatório de $2 + 2^2 + 2^3 + \cdots + 2^{x-1} + 2^x = 2^{x+1} - 2$,pois por exemplo, o somatório de $2 + 2^2 = 6$ em que $2^3 - 2 = 6$,o mesmo ocorre com de $2 + 2^2 + 2^3 = 14$ em que $2^4 - 2 = 14$ e assim por diante.

Com isso obtendo $F(n) = 2^{x+1} - 2 + 2^x F(n-x)$

Considerando que F(0)= 0, na função F(n-x) será igual a zero quando x=n;

Assim teria a seguinte expressão $F(n) = 2^{n+1} - 2 + 2^n 0$, ou seja $F(n) = 2^{n+1} - 2$

Tendo assim complexidade O(2^n)

Para quando a função recursiva tiver 1 instrução:

$\mathrm{F(n)} = 1 + 2F(n-1)$

$\mathrm{F(n)} = 3 + 2^2 F(n-2)$

$\mathrm{F(n)} = 7 + 2^3 F(n-1)$

Ou seja, $F(n) = 2^x - 1 + 2^x F(n-x)$

Considerando que F(0)= 0, na função F(n-x) será igual a zero quando x=n;

Assim teria a seguinte expressão $F(n) = 2^n - 1 + 2^n 0$, ou seja $F(n) = 2^n - 1$

Tendo assim complexidade O(2^n)

Para quando a função recursiva tiver 4 instruções:

$\mathrm{F(n)} = 4 + 2F(n-1)$

$\mathrm{F(n)} = 12 + 2^2F(n-2)$

$\mathrm{F(n)} = 28 + 2^3F(n-1)$

Ou seja, $F(n) = 2^{x+2} - 4 + 2^xF(n-x)$

Considerando que F(0)= 0, na função F(n-x) será igual a zero quando x=n;

Assim teria a seguinte expressão $F(n) = 2^{n+2} - 4 + 2^n0$, ou seja $F(n) = 2^{n+2} - 4$

Tendo assim complexidade O(2^n)

Para qualquer a função recursiva tiver a quantidade de instrução sendo múltipla de 2:

Para os casos em que se tem o número de instruções sendo múltiplo de 2, seguindo a mesma logica dos de cima, obterá sempre $F(n) = 2^{n+x} - y$, onde y = ao valor múltiplo de 2 e x igual ao valor do 2^x=y , assim:

1 instrução = $F(n) = 2^{n+0} - 1$ = $F(n) = 2^n - 1$

2 instruções = $F(n) = 2^{n+1} - 2$ = $F(n) = 2^{n+1} - 2$

4 instruções = $F(n) = 2^{n+2} - 4$ = $F(n) = 2^{n+2} - 4$

8 instruções = $F(n) = 2^{n+3} - 8$ = $F(n) = 2^{n+3} - 8$

Para qualquer função recursiva que chame ela mesma 2 vezes obterá O(2^n).

Cálculo complexidade Subset sum backtracking:

Imagem do código:

```java
public boolean existe_o_somatorio_arvore(Arvore_nó arvore_nó, int interação,String valores) {
    if(arvore_nó.getValor_somado() == this.getSomatorio()) {
        add_vet_valores(valores);
        return true;
    }
    else if(arvore_nó.getValor_restante()==0) {
        return false;
    }
    else {
        boolean ignora = existe_o_somatorio_arvore(new Arvore_nó(arvore_nó.getValor_restante()-Math.abs(this.getVetfloat()[interação]),arvore_nó.getValor_somado()),interação+1,valores);
        boolean considera = existe_o_somatorio_arvore(new Arvore_nó(arvore_nó.getValor_restante()-Math.abs(this.getVetfloat()[interação]),arvore_nó.getValor_somado()+this.getVetfloat()[interação]),
                interação+1,valores+this.getVetfloat()[interação]+" ");
        return ignora || considera;
    }
}
```

Obs.: No código acima não tem parada quando o valor somado no nó for maior que o valor do somatório, pois dessa forma ele encontrará os somatórios para valores negativos, como por exemplo 5,-3 e somatório 2 ele informara que possui o somatório 5-3=2.

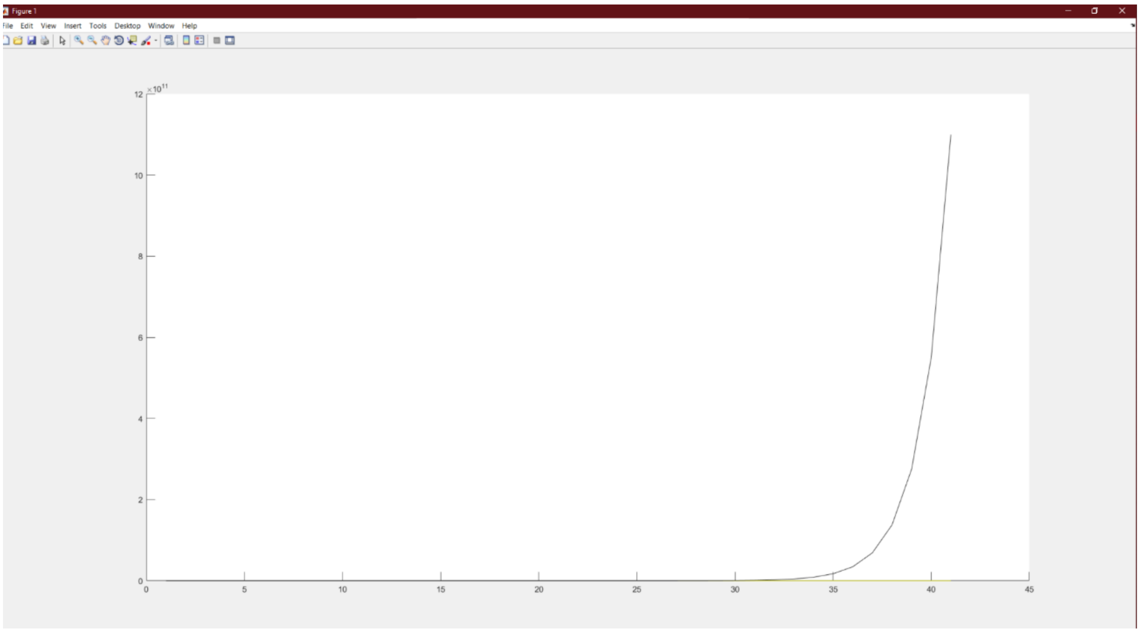
Este código terá a mesma complexidade do código Subset sum recursivo apresentado acima, pois se reparar o código e igual mudando pequenas coisas para que rode com a ideia do Backtrancking, como por exemplo, o objeto arvore que no recursivo não existe.

Demostrar a complexidade do algoritmo através da comparação com o gráfico das complexidades padrão:

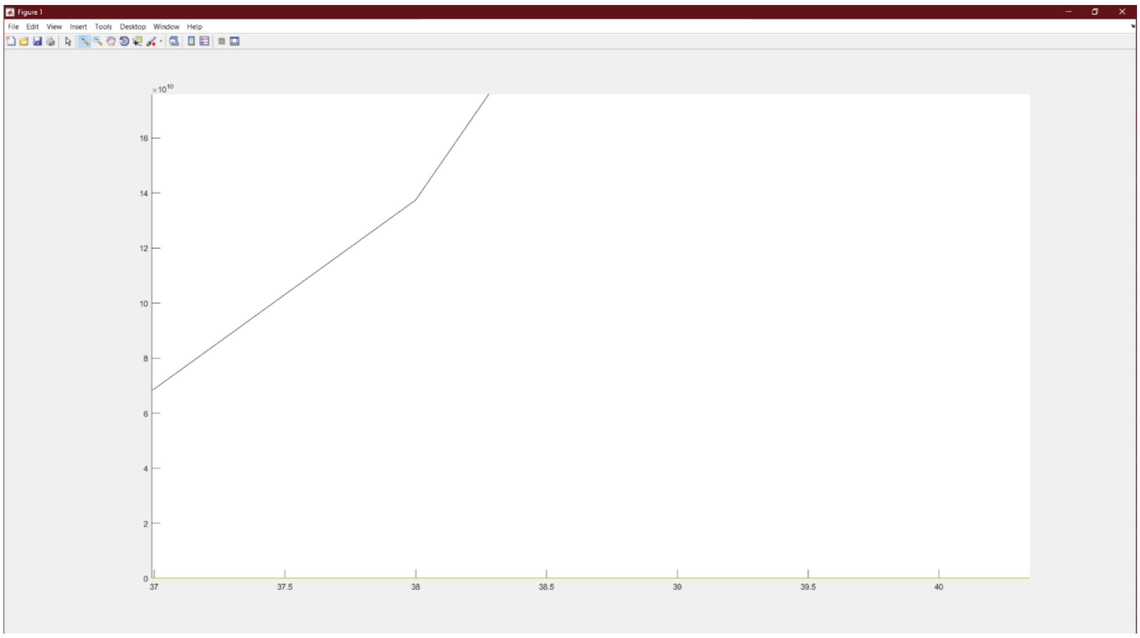
Utilizando o Matlab pode-se plotar o gráfico das complexidades padrões (O(1),O(logn),O(n),O(nlogn),O(n^2), O(n^3), O(2^n)) utilizando as operações da imagem a seguir:

```
>> n=0:1:40
hold on; plot(ones(1,40,'uint16'),'r'); plot(log(n),'g'); plot(n,'b'); plot(n.*log(n),'c');plot(n.^2,'m');plot(n.^3,'y');plot(2.^(n),'k')

n =

  Columns 1 through 27

     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

  Columns 28 through 41

    27    28    29    30    31    32    33    34    35    36    37    38    39    40

fx >> |
```

Assim os valores de O(1) estão em vermelho, os de O(logn) estão em verde, os de O(n) estão em azul, os de O(nlogn) estão em ciano, os de O(n^2) estão em magenta, os de O(n^3) estão em amarelo e os de O(2^n) estão em preto. Além disso os valores de n vão de 0 a 40, com isso obtendo o seguinte gráfico:



Ampliando a imagem:



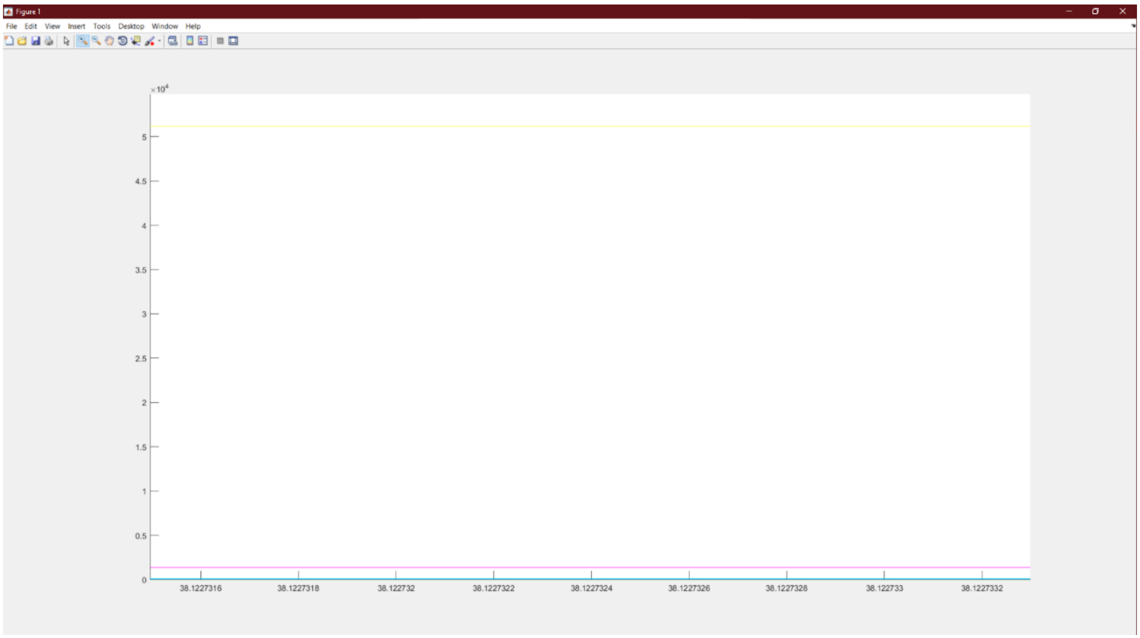


Figure 1
File Edit View Insert Tools Desktop Window Help
×10⁴
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
38.1227316
38.1227318
38.122732
38.1227322
38.1227324
38.1227326
38.1227328
38.122733
38.1227332

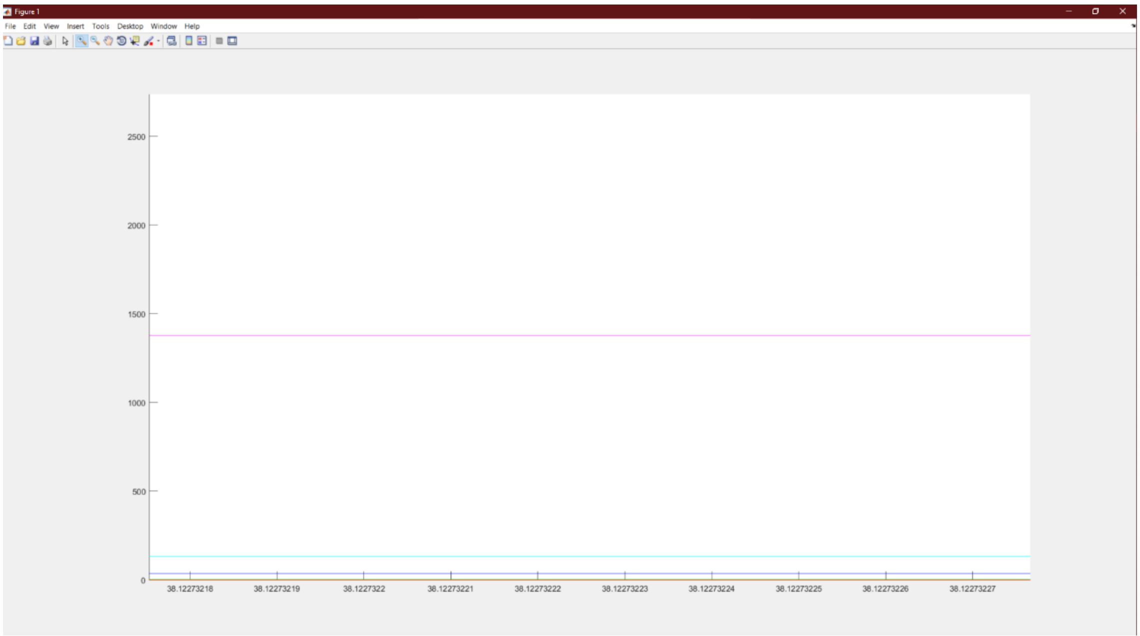
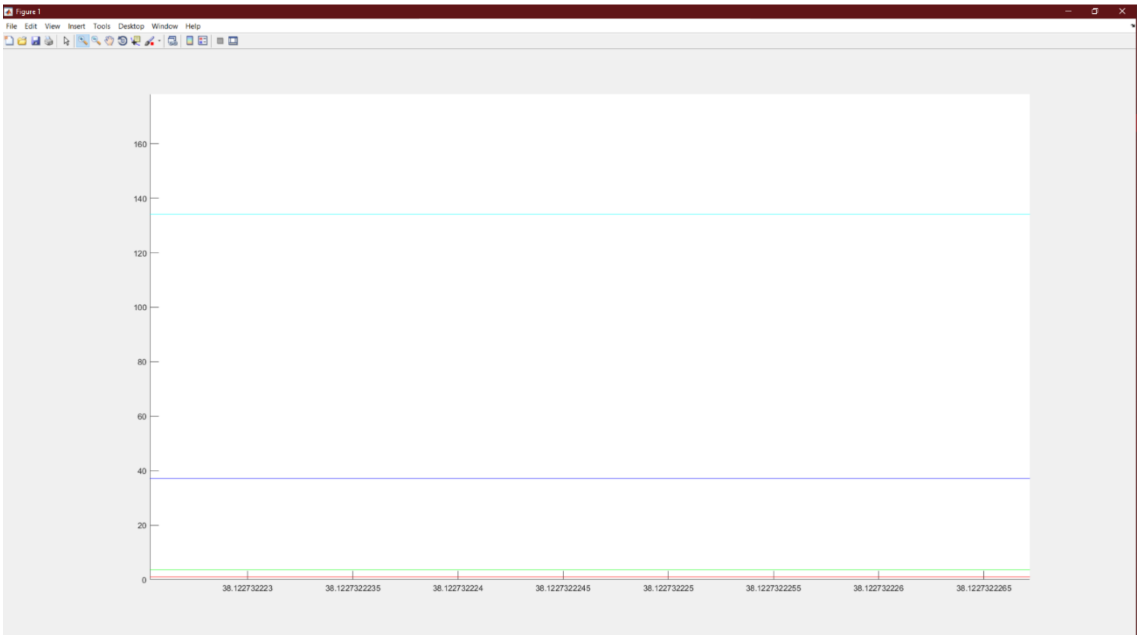


Figure 1
File Edit View Insert Tools Desktop Window Help
2500
2000
1500
1000
500
0
38.12273218
38.12273219
38.1227322
38.12273221
38.12273222
38.12273223
38.12273224
38.12273225
38.12273226
38.12273227



Com isso dá para ver que o O(2^n) e bem maior que o restos das complexidades e a ordem das complexidades que tem menor custo computacional quando os valores são muito grandes são (O(1),O(logn),O(n),O(nlogn),O(n^2), O(n^3), O(2^n)) ,assim pode-se ver que os algoritmos Subset sum recursivo e Backtracking mostrados gastam bem mais poder computacional que o dinâmico, quando os valores são maiores, sendo assim quanto maior o valor mais os algoritmos com complexidade o O(2^n) se distanciam dos de o O(n^2).