Alunos: Vinícius de Oliveira Guimarães RA: 802431
Vitor Enzo. RA: 802123

1 A programação dinâmica consiste na resolução de problemas a partir da solução de subproblemas. Entretanto, diferentemente da estratégia de dividir para conquistar, onde os subproblemas são disjuntos, na programação dinâmica os problemas maiores vão compartilhar os problemas maiores.

Dessa forma, a ideia é que os subproblemas sejam resolvidos apenas uma única vez, tendo seus resultados armazenados dentro de uma tabela para poder ser consultado, evitando retrabalho.

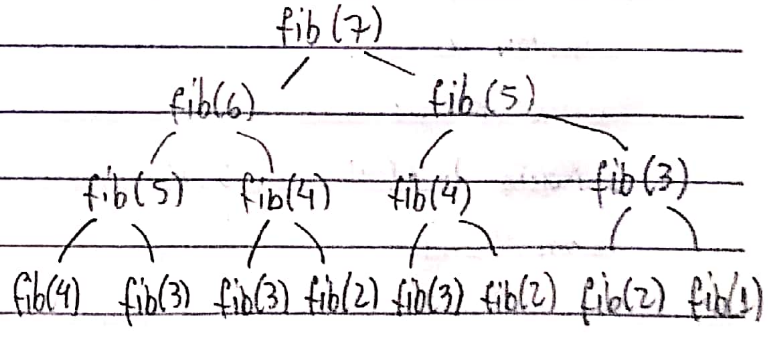
Assim, existem alguns passos para desenvolver um algoritmo usando programação dinâmica:

1- Definir a estrutura de uma solução ótima
2- Estabelecer, recursivamente, o valor de uma solução ótima
3- Calcular o valor de uma solução ótima
4- Construir uma solução ótima a partir da informação calculado.

2
```
Fibonacci(n){
  if n==0
    return 0
  if n==1
    return 1
  else
    return fibonacci(n-1)+fibonacci(n-2)
}
```

→ Calculando a complexidade observando a recorrência

[1] exemplo da árvore de recorrência:



[2] Observando a árvore de recorrência, podemos escrever a recorrência assim:

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

Quando n for para infinito, $T(n-1) \approx T(n-2)$. Ou seja:

$$T(n) = 2T(n-1) + O(1)$$
$$= 2.[2.T(n-2) + O(1)] + O(1)$$
$$= 2.[2.[2T(n-3) + O(1)] + O(1)] + O(1) = 2^3 T(n-3) + O(1)$$

Continuando até K:

$T(n) = 2^k T(n-k) + O(1)$

Como a condição de parada da recursão é $T(0)$, então $n-k=0 \rightarrow n=k$

Logo, temos:

$$T(n) = 2^n \underbrace{T(0)}_{O(1)} + O(1)$$

Como 2^n domina, então:

$T(n) = O(2^n)$, sendo assim uma complexidade exponencial, que é extremamente ruim (PROIBITIVO). Olhando na árvore de recursão podemos observar que muitos cálculos estão sendo realizados diversas vezes, sendo assim, provado pelos cálculos, ineficiente.

3

a) Abordagem Top-Down (Memorização)

Algoritmo da estratégia Top-Down:

```
M[0...n] = 0
Fib_M(n) {
    if(n == 0 || n == 1)
        return n
    if(M[n] == 0)
        M[n]=Fib_M(n-1) + Fib_M(n-2)
return M[n]
```

Observando o código, podemos observar que o array de valores calculado é iniciado com o valor 0 para todas as posições e que o mesmo é preenchido conforme os cálculos são feitos, evitando retrabalho.

Assim, a complexidade é diminuída consideravelmente por conta do armazenamento do resultado dos subproblemas

b) Abordagem bottom-up (recursão)

Algoritmo da estratégia bottom-up

```
Fib_B(n) {
    if (n==0)
        return 0;
    else {
        previous = 0
        current = 1
        for (i=1 to n) {
            new = previous + current
            previous = current
            current = new
        }
        return current
    }
```

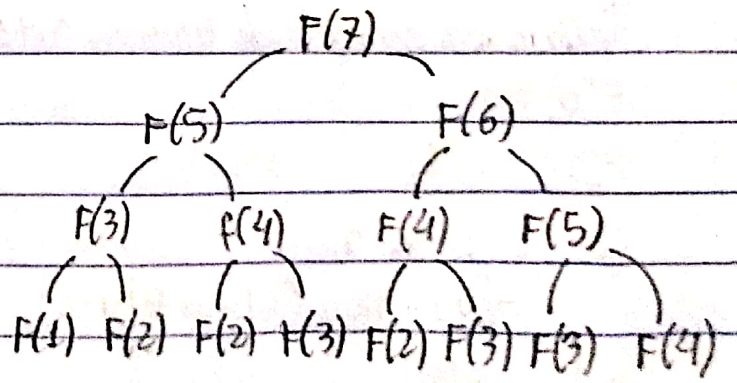
Observando a função Fib_B, observa-se que a eficiência do algoritmo é O(n), sendo muito melhor do que a forma com eficiência exponencial.

A eficiência desse algoritmo ficou muito melhor pois na forma iterativa, é possível calcular a sequência de Fibonacci modificando os valores de previous e current conforme os cálculos são realizados, bastando um for de i=1 até n.

4) Para o problema da contagem de cédulas, podemos criar o seguinte código para resolver o problema:

```
F(n) {
    if n==0
        return 0
    else {
        if n==1
            return c[1]
        else
            return max(c[n]+F(n-2), F(n-1))
    }
}
```

Observando a árvore de recorrência:



Nota-se que há muita repetição de cálculo, sendo ineficiente. Vamos calcular a complexidade desse algoritmo:

Dado a sequência de recursão sendo $T(n) = T(n-1) + T(n-2) + O(1)$, então:
Quando n vai para o infinito $T(n-1) \approx T(n-2)$
Logo, $T(n) = 2T(n-1) + O(1)$

$$= 2[2T(n-2) + O(1)] + O(1)$$
$$= 2\cdot[2\cdot[2\cdot T(n-3) + O(1)] + O(1)] + O(1)$$
$$= 2^3 \cdot T(n-3) + O(1)$$

Continuando K:

$T(n) = 2^k \cdot T(n-k) + O(1)$, como a condição de parada é até $T(0)$:

$$T(n) = 2^k \cdot \underbrace{T(0)}_{O(1)} + O(1)$$

$T(n) = O(2^K)$, que é exponencial (Extremamente ruim)

Portanto, como vários cálculos precisam ser refeitos diversas vezes (observando árvore de recursão), conclui-se que $O(2^K)$ é muito ruim.

⑤ Abordagem bottom-up

Podemos escrever o algoritmo para solucionar o problema da sequência de cédulas como sendo:

```
Cedulas (n){
    Definir um array F de tamanho n+1
    F[0] = 0
    F[1] = C[1]          1 2 3 4
    for i=2 to n
        F[i] = max (C[i]+F[i-2], F[i-1])
    return F[n]
}
```

Analisando o código, percebemos que o mesmo possui complexidade $O(n)$, uma vez que há um laço que percorre $n-1$ vezes, sendo da ordem $O(n)$.

Assim, como a complexidade é $O(n)$, seu algoritmo é muito mais eficiente do que o exponencial.

⑥ $C = [2, 5, 5, 2, 10, 50, 100, 50, 20, 20, 50, 100]$

Inicialmente temos que $F[0] = 0$ e $F[1] = C_1 = 2$, Assim temos:

Valor de i	Valor para F[i]
$i=2$	$F[2] = \max(C[2]+F[2-2], F[2-1]) = \max(5+0, 2) = 5$
$i=3$	$F[3] = \max(C[3]+F[3-2], F[3-1]) = \max(5+2, 5) = 7$
$i=4$	$F[4] = \max(C[4]+F[4-2], F[4-1]) = \max(2+5, 7) = 7$
$i=5$	$F[5] = \max(C[5]+F[5-2], F[5-1]) = \max(10+7, 7) = 17$
$i=6$	$F[6] = \max(C[6]+F[6-2], F[6-1]) = \max(50+7, 17) = 57$
$i=7$	$F[7] = \max(C[7]+F[7-2], F[7-1]) = \max(100+17, 57) = 117$
$i=8$	$F[8] = \max(C[8]+F[8-2], F[8-1]) = \max(50+57, 117) = 117$
$i=9$	$F[9] = \max(C[9]+F[9-2], F[9-1]) = \max(20+117, 117) = 137$
$i=10$	$F[10] = \max(C[10]+F[10-2], F[10-1]) = \max(20+117, 137) = 137$
$i=11$	$F[11] = \max(C[11]+F[11-2], F[11-1]) = \max(50+137, 137) = 187$
$i=12$	$F[12] = \max(C[12]+F[12-2], F[12-1]) = \max(100+137, 187) = 237$

Assim o valor máximo de dinheiro que pode ser coletado sem que 2 cédulas vizinhas sejam coletadas é 237.

AA4 – Programação Dinâmica

Questão 07)

```
Robo-coletor(c){
    let F[0..n, 0..m] be an array
    F[1,1] = C[1,1]
    for j = 2 to m
        F[1,j] = F[1,j-1] + C[1,j]
    for i = 2 to n {
        F[i,1] = F[i-1,1] + C[i,1]
        for j = 2 to m
            F[i,j] = max(F[i-1,j], F[i,j-1]) + C[i,j]
    }
    return F[n,m]
}
```

É fácil notar que a complexidade do primeiro e do segundo loop é, respectivamente, $O(m)$ e $O(n \cdot m)$. Portanto a complexidade do algoritmo é $O(m) + O(n \cdot m)$ e como $O(n \cdot m) > O(m)$ então a complexidade equivale à $O(n \cdot m)$.

Percebe-se, portanto, que o algoritmo Bottom-up é muito mais eficiente que a complexidade $O(2^{n+m})$ do método recursivo.

Questão 08)

F[nxm]

0	0	1	1	2	2
1	2	2	3	3	3
1	3	3	3	4	4
1	3	4	4	4	5
1	4	5	5	6	6
1	4	6	6	7	7

C[nxm]

		•		•	
•	•		•		
	•			•	
		•			•
	•	•		•	
		•		•	

Questão 09)

```
Cut_Rod_Mem(p,n) {
    let r[0 n] be an array
    for i=0 to n
        r[i] = -inf
    return Cut_Rod_Mem_Aux(p,n,r)
}

Cut_Rod_Mem_Aux(p,n,r){
    if r[n] >= 0
        return r[n]
    if n == 0
        q = 0
    else {
        q = -inf
        for i=0 to n
            q = max{q, p[i] + Cut_Rod_Mem_Aux(p,n-i,r)}
    }
    r[n] = q
    return q
}
```

A função Cut_Rod_Mem_Aux resolve o problema para os tamanhos de 0, 1, ..., n. Porém, cada problema é iterado em um for de tamanho n. $O(n^2)$ é sua complexidade.

Esse algoritmo em PD é muito mais eficiente que sua forma recursiva, a qual possui complexidade $O(2^n)$

Questão 10) Resolva para $n = 6$

l_i	1	2	3	4	5	6
p_i	2	6	9	10	12	16

$r[0: \underline{2}, \underline{6}, \underline{9}, \underline{12}, \underline{15}, \underline{18}]$
   0 1 2 3 4 5 6

$q =$

$j=1, i=1$ $q = \max\{-\infty, 2\} = 2$

$j=2, i=1$ $q = \max\{-\infty, 2+2\} = 4$

$i=2$ $q = \max\{4, 6+0\} = 6$

$j=3, i=1$ $q = \max\{-\infty, 2+6\} = 8$

$i=2$ $q = \max\{8, 6+2\} = 8$

$i=3$ $q = \max\{8, 9+0\} = 9$

$j=4, i=1$ $q = \max\{-\infty, 2+9\} = 11$

$i=2$ $q = \max\{11, 6+6\} = 12$

$i=3$ $q = \max\{12, 9+2\} = 12$

$i=4$ $q = \max\{12, 10+0\} = 12$

$j=5, i=1$ $q = \max\{-\infty, 2+12\} = 14$

$i=2$ $q = \max\{14, 6+9\} = 15$

$i=3$ $q = \max\{15, 9+6\} = 15$

$i=4$ $q = \max\{15, 10+2\} = 15$

$i=5$ $q = \max\{15, 12+0\} = 15$

$j=6, i=1$ $q = \max\{-\infty, 2+15\} = 17$

$i=2$ $q = \max\{17, 6+12\} = 18$

$i=3$ $q = \max\{18, 9+9\} = 18$

$i=4$ $q = \max\{18, 10+6\} = 18$

$i=5$ $q = \max\{18, 12+2\} = 18$

$i=6$ $q = \max\{18, 16+0\} = 18$