SIN5013 - Análise de Algoritmos e Estruturas de Dados

Algoritmos gulosos e tentativa e erro

Prof. Flávio Luiz Coutinho

Algoritmos gulosos e tentativa e erro

- Duas abordagens distintas para resolução de problemas.
- Cada uma com características distintas:
    - desempenho
    - “qualidade” da solução

- Problemas para os quais as soluções são construídas como uma sequência de passos, e para cada passo temos várias opções a serem tomadas.

- Exemplo: problema do troco

Problema de troco

- O problema: compor o valor n usando a menor quantidade de cédulas/moedas dentre aquelas disponíveis.

- Exemplo:
    - n = 183
    - v = { 50, 20, 10, 5, 2, 1 }
    - solução: 50, 50, 50, 20, 10, 2, 1

- Cada passo define a próxima nota a ser incluída na solução.

Problema de troco

- Há diversas opções possíveis para executar cada passo (todas as notas disponíveis abaixo do valor remanescente para compor o valor total n).

Problema de troco

- Há diversas opções possíveis para executar cada passo (todas as notas disponíveis abaixo do valor remanescente para compor o valor total n).
- Mas intuitivamente parece razoável sempre priorizar o uso da maior nota disponível.

Problema de troco

- Há diversas opções possíveis para executar cada passo (todas as notas disponíveis abaixo do valor remanescente para compor o valor total n).
- Mas intuitivamente parece razoável sempre priorizar o uso da maior nota disponível.
- Uso das maiores notas reduzirá a quantidade de notas necessárias para compor a solução.

Problema de troco

- Há diversas opções possíveis para executar cada passo (todas as notas disponíveis abaixo do valor remanescente para compor o valor total n).
- Mas intuitivamente parece razoável sempre priorizar o uso da maior nota disponível.
- Uso das maiores notas reduzirá a quantidade de notas necessárias para compor a solução.

- O algoritmo apresentado a seguir implementa esta ideia...

Problema de troco (Algoritmo)

```
Solucao troco(int n, int * v, int k){

    Solucao solucao = nova_solucao();

    for(int i = 0; i < k; i++){

        while(n >= v[i]){

            n = n - v[i];
            solucao.adiciona(v[i]);
        }
    }

    return solucao;
}
```

Problema de troco (Algoritmo)

```
Solucao troco(int n, int * v, int k){

    Solucao solucao = nova_solucao();

    for(int i = 0; i < k; i++){

        while(n >= v[i]){

            n = n - v[i];
            solucao.adiciona(v[i]);
        }
    }

    return solucao;
}
```

O(n)

Problema de troco (Algoritmo)

```
Solucao troco(int n, int * v, int k){

    Solucao solucao = nova_solucao();

    for(int i = 0; i < k; i++){

        while(n >= v[i]){

            n = n - v[i];
            solucao.adiciona(v[i]);
        }
    }

    return solucao;
}
```

$O(n) * k$

Problema de troco (Algoritmo)

```
Solucao troco(int n, int * v, int k){

    Solucao solucao = nova_solucao();

    for(int i = 0; i < k; i++){

        while(n >= v[i]){

            n = n - v[i];
            solucao.adiciona(v[i]);
        }
    }

    return solucao;
}
```

$O(n) * \Theta(1)$

Problema de troco (Algoritmo)

```
Solucao troco(int n, int * v, int k){

    Solucao solucao = nova_solucao();

    for(int i = 0; i < k; i++){

        while(n >= v[i]){

            n = n - v[i];
            solucao.adiciona(v[i]);
        }
    }

    return solucao;
}
```

$$O(n) * \Theta(1) = O(n)$$

Problema de troco (Algoritmo)

```
Solucao troco(int n, int * v, int k){

    Solucao solucao = nova_solucao();

    for(int i = 0; i < k; i++){

        while(n >= v[i]){

            n = n - v[i];
            solucao.adiciona(v[i]);
        }
    }

    return solucao;
}
```

$$O(n) * \Theta(1) = O(n)$$

Considerando apenas a iteração do for em que i = 0, o bloco interno ao while irá executar **n / v[0]** vezes. Considerando todas as iterações, temos pelo menos **n / v[0]** execuções do bloco interno. Desta forma, o algoritmo também é **Ω(n)**.

Problema de troco (Algoritmo)

E portanto, o algoritmo tem complexidade **Θ(n)**

```
Solucao troco(int n, int * v, int k){

    Solucao solucao = nova_solucao();

    for(int i = 0; i < k; i++){

        while(n >= v[i]){

            n = n - v[i];
            solucao.adiciona(v[i]);
        }
    }

    return solucao;
}
```

$$O(n) * \Theta(1) = O(n)$$

Considerando apenas a iteração do for em que i = 0, o bloco interno ao while irá executar **n / v[0]** vezes. Considerando todas as iterações, temos pelo menos **n / v[0]** execuções do bloco interno. Desta forma, o algoritmo também é **Ω(n)**.

Problema de troco (análise)

Sem prever exatamente quantas vezes o bloco interno ao while irá executar no total para resolver um problema de tamanho n, conseguimos deduzir que este algoritmo é $\Theta(n)$.

Problema de troco (análise)

Sem prever exatamente quantas vezes o bloco interno ao while irá executar no total para resolver um problema de tamanho n, conseguimos deduzir que este algoritmo é $\Theta(n)$.

Antes de prosseguir uma observação: vale notar que se o objetivo fosse determinar “apenas” quantas notas de cada tipo seriam usadas (ao invés gerar a sequência de todas as notas necessárias), seria possível desenvolver um algoritmo com complexidade $\Theta(1)$.

Problema de troco

Mas será que o algoritmo funciona corretamente?

Problema de troco

Mas será que o algoritmo funciona corretamente?

- $n = 80$, $v = \{ 50, 20 \}$

Problema de troco

Mas será que o algoritmo funciona corretamente?

- $n = 80$, $v = \{ 50, 20 \}$ solução: 50, 20, ?

Problema de troco

Mas será que o algoritmo funciona corretamente?

- n = 80, v = { 50, 20 }   solução: 50, 20, ?
- n = 6,   v = {4, 3, 1 }

Problema de troco

Mas será que o algoritmo funciona corretamente?

- n = 80, v = { 50, 20 } solução: 50, 20, ?
- n = 6, v = {4, 3, 1 } solução: 4, 1, 1

Problema de troco

Mas será que o algoritmo funciona corretamente?

- n = 80, v = { 50, 20 } solução: 50, 20, ?
- n = 6, v = {4, 3, 1 } solução: 4, 1, 1

Dependendo do valor de n e do conjunto de notas o algoritmo proposto **nem sempre consegue dar uma resposta, ou dar a resposta "ótima".**

Problema de troco

Mas será que o algoritmo funciona corretamente?

- n = 80, v = { 50, 20 } solução: 50, 20, ?
- n = 6, v = {4, 3, 1 } solução: 4, 1, 1

Dependendo do valor de n e do conjunto de notas o algoritmo proposto **nem sempre consegue dar uma resposta, ou dar a resposta "ótima"**.

Intuição usada para dar um passo na construção da solução, sempre priorizando a nota de maior valor, e sem reconsiderar as escolhas já feitas, **não garante soluções ótimas** em 100% dos casos.

Problema de troco

Como garantir uma solução ótima???

Problema de troco

Como garantir uma solução ótima???

Testando todas as possíveis formas de compor o valor do troco, e escolhendo aquela que emprega a menor quantidade de notas: abordagem por tentativa e erro.

Problema de troco

Como garantir uma solução ótima???

Testando todas as possíveis formas de compor o valor do troco, e escolhendo aquela que emprega a menor quantidade de notas: abordagem por tentativa e erro.

Contraponto com a abordagem usada no algoritmo recém apresentado, que é uma abordagem gulosa (*greedy*).

Problema de troco

Como garantir uma solução ótima???

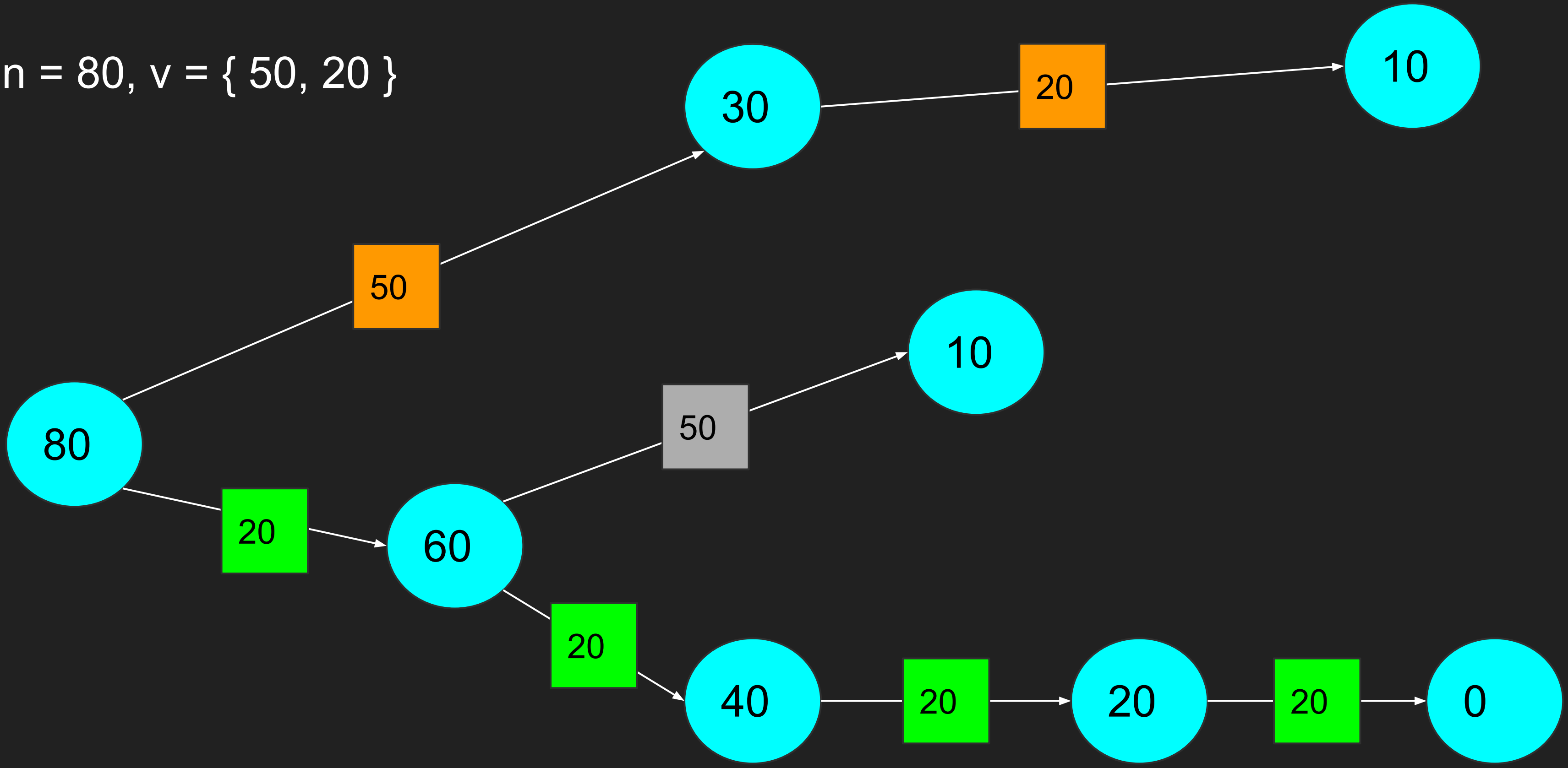
Testando todas as possíveis formas de compor o valor do troco, e escolhendo aquela que emprega a menor quantidade de notas: abordagem por tentativa e erro.

Contraponto com a abordagem usada no algoritmo recém apresentado, que é uma abordagem gulosa (*greedy*).

Diagrama para os dois exemplos em que o uso da abordagem gulosa não leva a soluções ótimas.

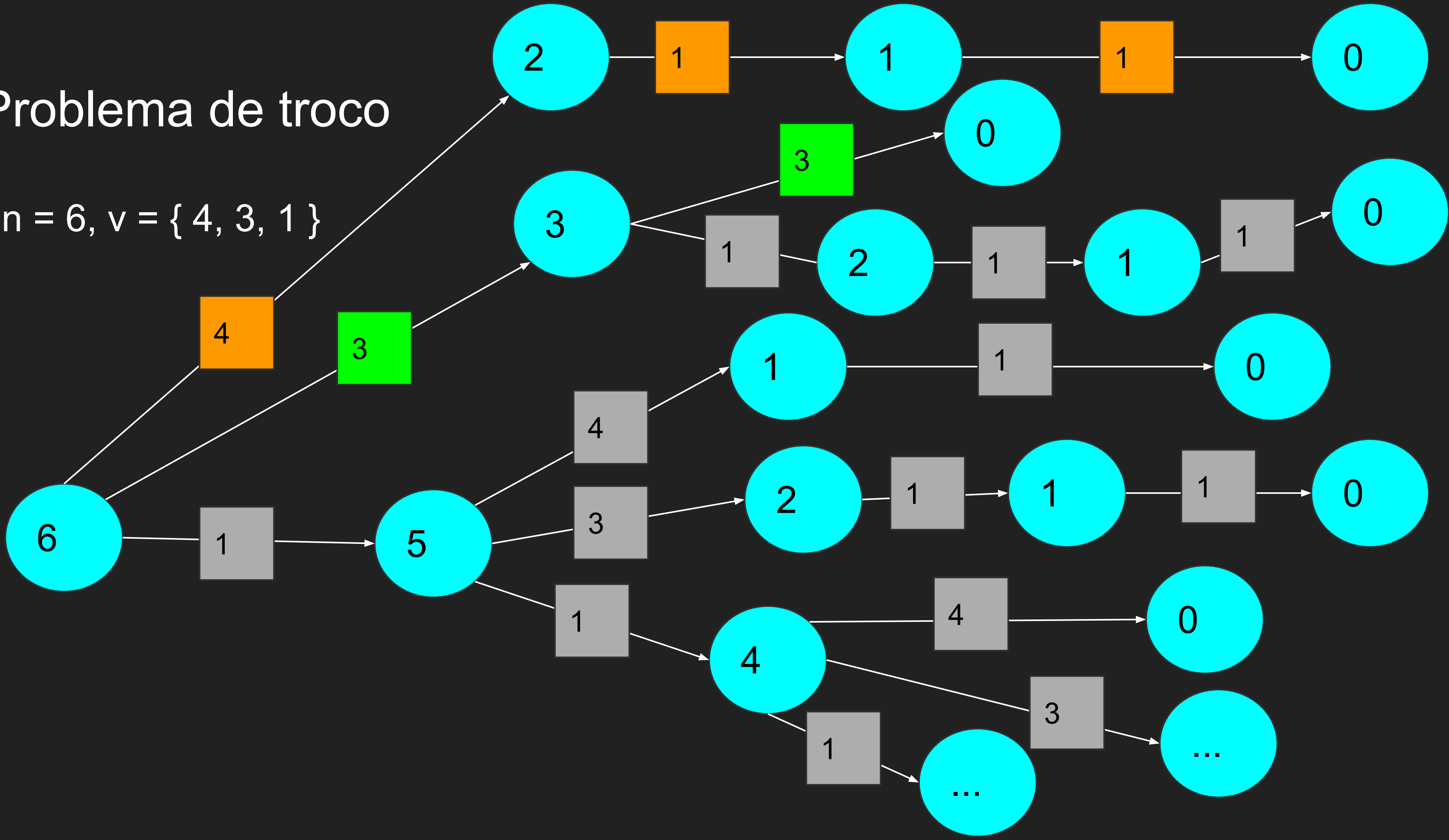
Problema de troco

$n = 80, v = \{ 50, 20 \}$



Problema de troco

n = 6, v = { 4, 3, 1 }



Esboço de um algoritmo “Tentativa e Erro”

```
estado_atual = estado_inicial
while(houver opções não exploradas no estado_atual){
    if(estado_atual é solução candidata para o problema){
        verifica se é a melhor solução conhecida até o momento
    }

    if(é possível ir a um nó filho (novo estado) a partir do estado atual){
        vá até o nó filho (atualiza estado atual), registrando o passo dado
    }
    else { desfaz a última ação feita e retorna-se ao estado anterior }
}
```

Esboço de um algoritmo “Tentativa e Erro”

```
estado_atual = estado_inicial
while(houver opções não exploradas no estado_atual){
    if(estado_atual é solução candidata para o problema){
        verifica se é a melhor solução conhecida até o momento
    }

    if(é possível ir a um nó filho (novo estado) a partir do estado atual){
        vá até o nó filho (atualiza estado atual),  registrando o passo dado
    }
    else { desfaz a última ação feita e retorna-se ao estado anterior }
}
```

Esboço de um algoritmo “Tentativa e Erro”

```
estado_atual = estado_inicial
while(houver opções não exploradas no estado_atual){
    if(estado_atual é solução candidata para o problema){
        verifica se é a melhor solução conhecida até o momento
    }

    if(é possível ir a um nó filho (novo estado) a partir do estado atual){
        vá até o nó filho (atualiza estado atual),  registrando o passo dado
    }
    else { desfaz a última ação feita  e retorna-se ao estado anterior }
}
```

Algoritmo Tentativa e Erro

Pontos importantes para gerenciar a exploração da árvore de possibilidades:

- manter registro dos passos executados que levaram ao estado atual.
- ser capaz de voltar um passo atrás, quando não há mais como prosseguir:
  - BACKTRACKING!

Como fazer uma implementação que garante estes aspectos?

- usando uma PILHA.
- elemento da pilha: estado corrente e último passo já testado a partir do estado corrente.

Algoritmo Tentativa e Erro

Pontos importantes para gerenciar a exploração da árvore de possibilidades:

- manter registro dos passos executados que levaram ao estado atual.
- ser capaz de voltar um passo atrás, quando não há mais como prosseguir:
    - BACKTRACKING!

Como fazer uma implementação que garante estes aspectos?

- usando uma PILHA.
- elemento da pilha: estado corrente e último passo já testado a partir do estado corrente.
- recursão nos dá essa pilha “de graça” (a própria pilha de chamadas).

Algoritmo Tentativa e Erro

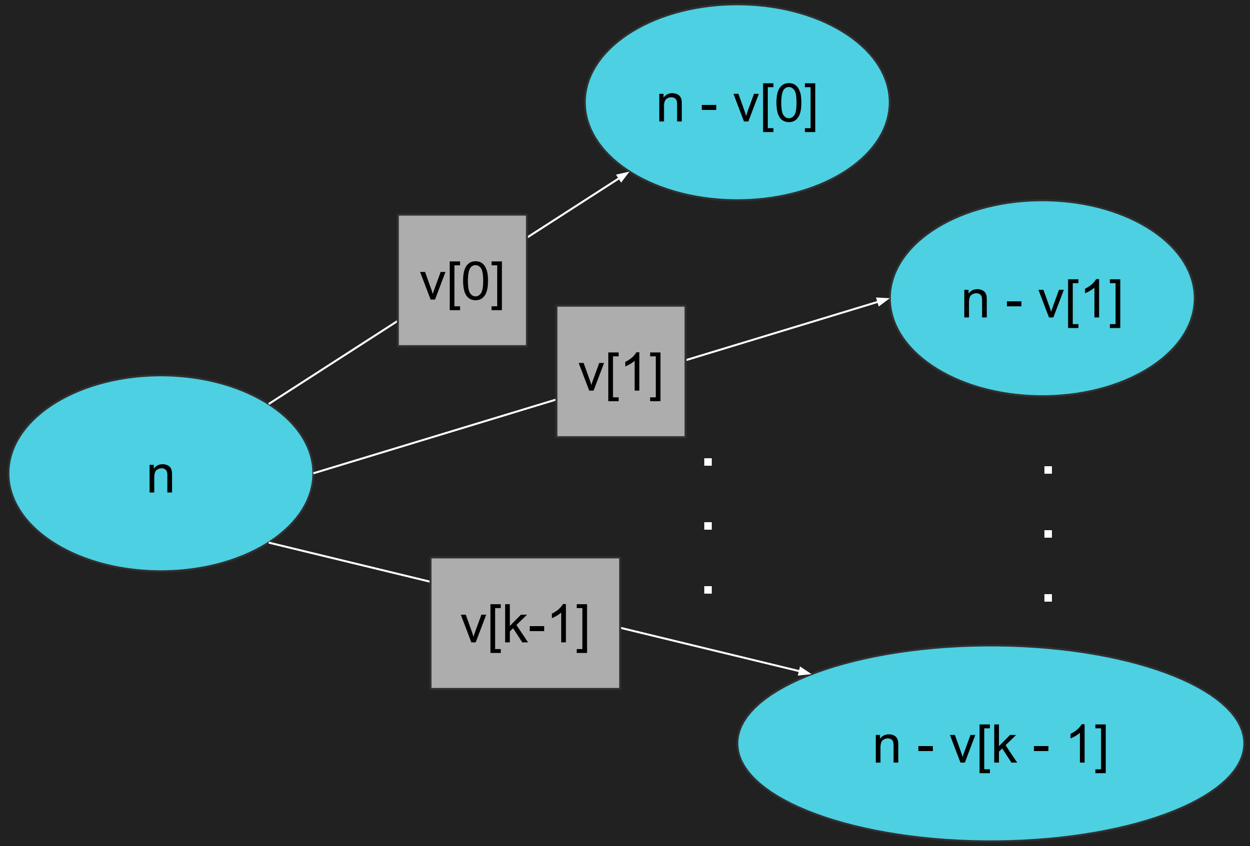
Pontos importantes para gerenciar a exploração da árvore de possibilidades:

- manter registro dos passos executados que levaram ao estado atual.
- ser capaz de voltar um passo atrás, quando não há mais como prosseguir:
  - BACKTRACKING!

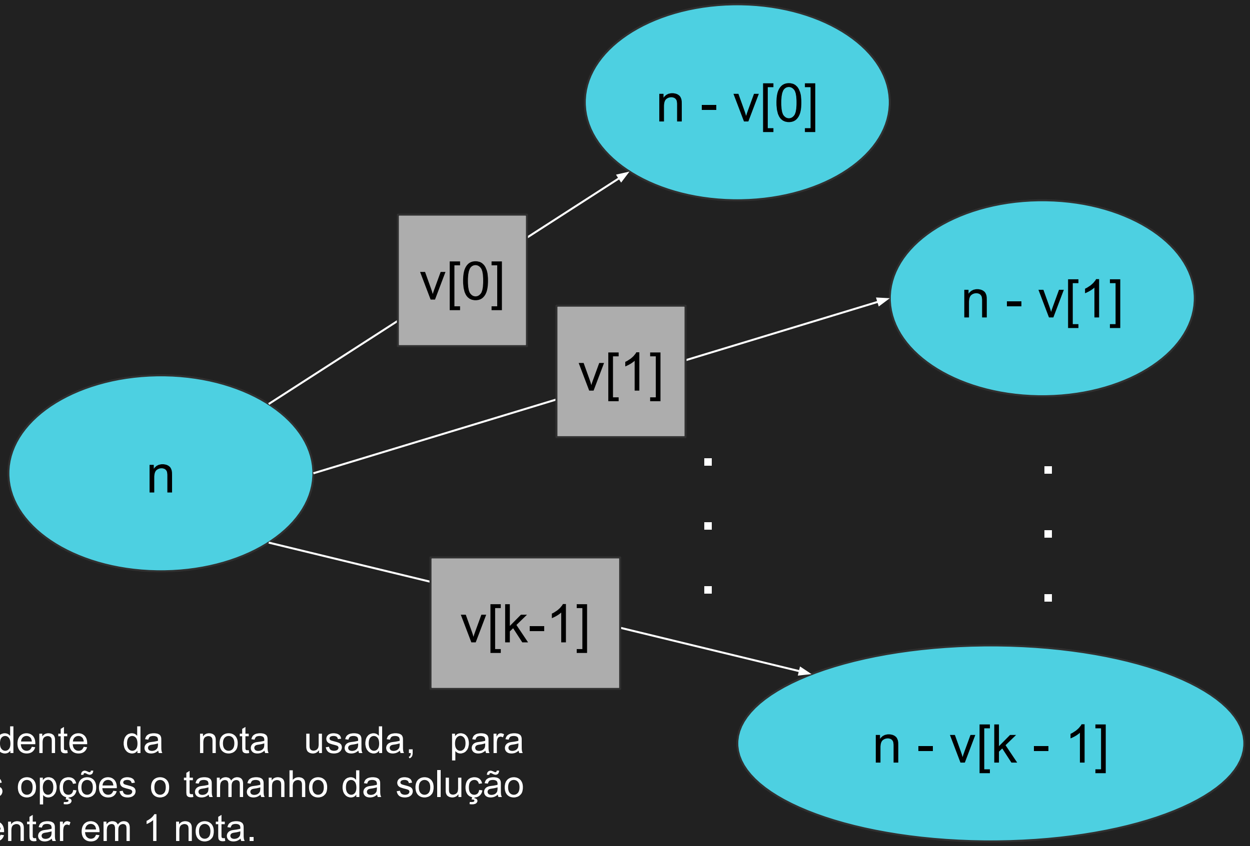
Como fazer uma implementação que garante estes aspectos?

- usando uma PILHA.
- elemento da pilha: estado corrente e último passo já testado a partir do estado corrente
- recursão nos dá essa pilha “de graça” (a própria pilha de chamadas).
- costuma ser mais fácil implementar um algoritmo “tentativa e erro” de forma recursiva.

Solução “tentativa e erro” para o problema do troco

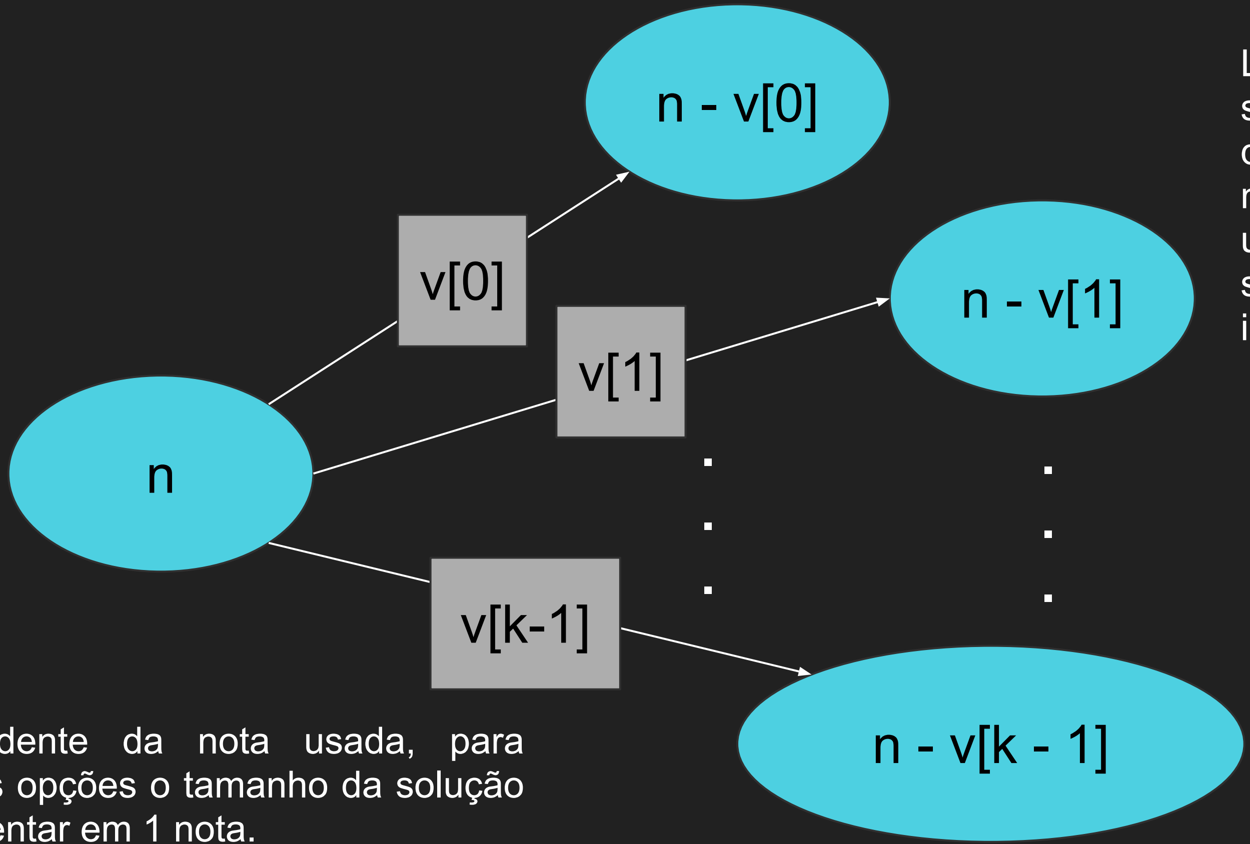


Solução “tentativa e erro” para o problema do troco



Independente da nota usada, para todas as opções o tamanho da solução irá aumentar em 1 nota.

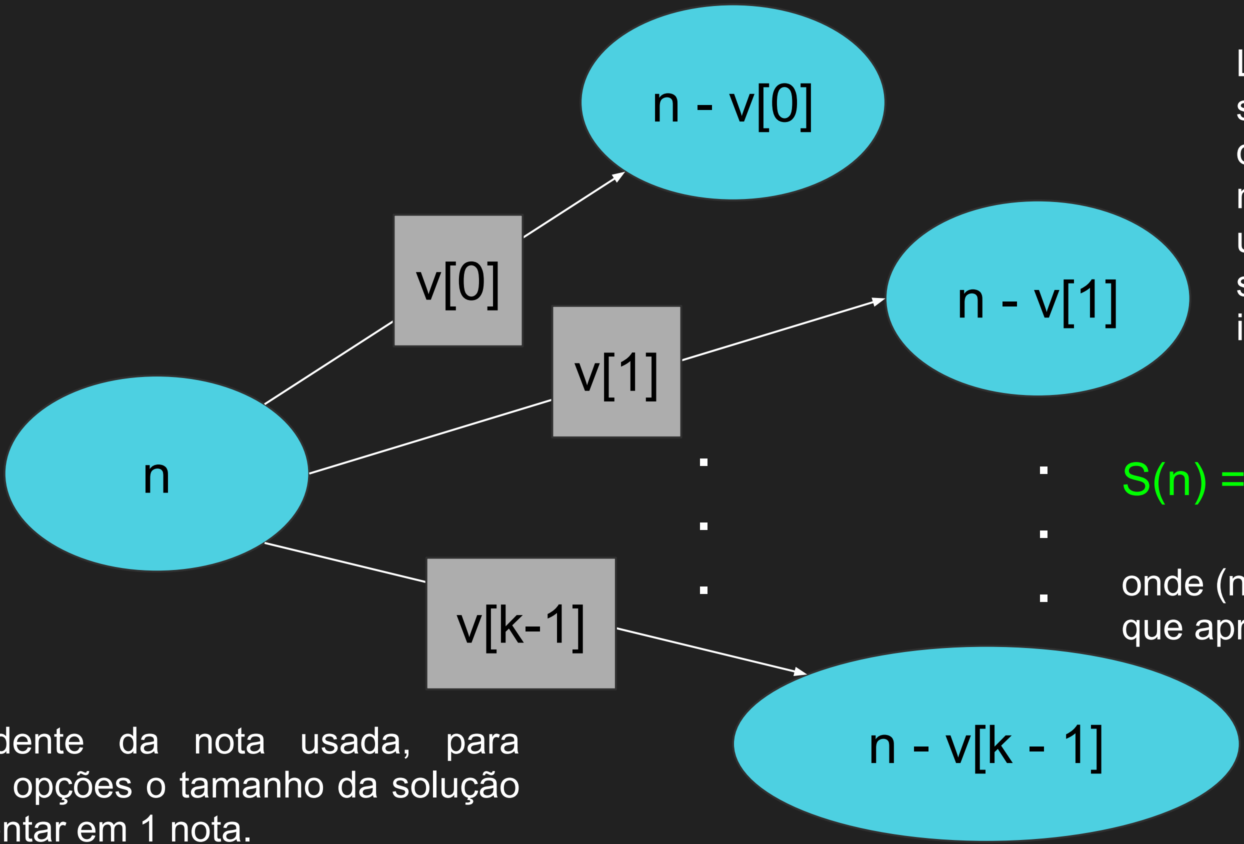
Solução “tentativa e erro” para o problema do troco



Logo, a solução do subproblema (n - v[i]) que usar o menor número de notas, será usada para compor a solução do problema inicial.

Independente da nota usada, para todas as opções o tamanho da solução irá aumentar em 1 nota.

Solução "tentativa e erro" para o problema do troco



Logo, a solução do subproblema (n - v[i]) que usar o menor número de notas, será usada para compor a solução do problema inicial.

$S(n) = S(n - v[i]) \cup \{ v[i] \}$

onde (n - v[i]) é o subproblema que apresenta a melhor solução

Independente da nota usada, para todas as opções o tamanho da solução irá aumentar em 1 nota.

Solução “tentativa e erro” para o problema do troco - implementação

```
Solucao tentativa_e_erro(int n, int * v, int k){

    if(n == 0) return nova_solucao();

    Solucao melhor = NULL;
    int valor = -1;

    for(int i = 0; i < k; i++){

      if(n - v[i] >= 0){

           Solucao solucao = tentativa_e_erro(n - v[i], v, k);

           if(solucao && (melhor == NULL || solucao.tamanho() < melhor.tamanho())){

                melhor = solucao; valor = v[i];
           }
      }
    }

    if(melhor) melhor.adiciona(valor);
    return melhor;
}
```

Algoritmo Tentativa e Erro - Análise

```
T(n)   = c1                                     [ para n = 0 ]
       = Σ [ T(n - v[i]) ] + c2                 [ para n > 0 ]

         com i indo de 0 até k - 1, onde k = |v|
```

Algoritmo Tentativa e Erro - Análise

```
T(n)   = c1                                    [ para n = 0 ]
       = Σ [ T(n - v[i]) ] + c2                [ para n > 0 ]

         com i indo de 0 até k - 1, onde k = |v|
```

Não é uma recorrência simples de resolver... :(

Algoritmo Tentativa e Erro - Análise

```
T(n)   = c1                                 [ para n = 0 ]
       = Σ [ T(n - v[i]) ] + c2             [ para n > 0 ]

          com i indo de 0 até k - 1, onde k = |v|
```

Não é uma recorrência simples de resolver... :(

Tamanho dos k subproblemas são variáveis...

Algoritmo Tentativa e Erro - Análise

$$T(n) = c_1 \qquad [\text{ para } n = 0 \text{ }]$$

$$= \Sigma \text{ [} T(n - v[i]) \text{] } + c_2 \qquad [\text{ para } n > 0 \text{ }]$$

com i indo de 0 até k - 1, onde k = |v|

Não é uma recorrência simples de resolver... :(

Tamanho dos k subproblemas são variáveis...

Vamos tentar simplificar um pouco as coisas.

Algoritmo Tentativa e Erro - Análise

$$T(n) = c_1 \qquad [\text{ para } n = 0 \text{ }]$$
$$= \Sigma \text{ [} T(n - v[i]) \text{] } + c_2 \qquad [\text{ para } n > 0 \text{ }]$$

com i indo de 0 até k - 1, onde k = |v|

Vamos definir uma nova recorrência T’(n) que sirva como limite inferior da recorrência T(n), ou seja, T(n) >= T’(n)

$$T'(n) = k * T'(n - v_0) + c_2 \qquad \text{onde } v_0 = v[0]$$

Algoritmo Tentativa e Erro - Análise

$T(n) = c_1$ [para n = 0]

$= \Sigma$ [$T(n - v[i])$] $+ c_2$ [para n > 0]

com i indo de 0 até k - 1, onde k = |v|

Vamos definir uma nova recorrência T'(n) que sirva como limite inferior da recorrência T(n), ou seja, T(n) >= T'(n)

$T'(n) = k * T'(n - v_0) + c_2$ onde $v_0 = v[0]$

$(n - v_0)$ é o **menor** dos subproblemas

Algoritmo Tentativa e Erro - Análise

(0) $T'(n)$

(1) $T'(n) = k * T'(n - v_0) + c_2$

(2) $T'(n) = k * (k * T'(n - 2v_0) + c_2) + c_2$

(2) $T'(n) = k^2 * T'(n - 2v_0) + kc_2 + c_2$

(3) $T'(n) = k^2 * (k * T'(n - 3v_0) + c_2) + kc_2 + c_2$

(3) $T'(n) = k^3 * T'(n - 3v_0) + c_2(k^2 + k^1 + k^0)$

...

(i) $T'(n) = k^i * T'(n - iv_0) + c_2 \Sigma(k^j)$ $\quad 0 <= j < i$

Algoritmo Tentativa e Erro - Análise

(i) $T'(n) = k^{i} * T'(n - iv_0) + c_2 \Sigma k^{j}$ $0 <= j < i$

caso base ocorre na expansão (n/v_0)

...

(n/v_0) $T'(n) = k^{(n / v0)} * T'(0) + c_2 \Sigma(k^{j})$ $0 <= j < n/v_0$

(n/v_0) $T'(n) = [k^{(1 / v0)}]^{n} * c_1 + ...$

(n/v_0) $T'(n) = c_3^{n} * c_1 + ...$ $c_3 = k^{(1 / v0)}$

(n/v_0) $T'(n) >= c_3^{n} * c_1$

(n/v_0) $T'(n) = \Omega(c_3^{n})$

Algoritmo Tentativa e Erro - Análise

$T'(n) = \Omega(c_3^{\,n})$

T'(n) é limitado inferiormente por uma função exponencial!

T(n) também é limitado inferiormente por uma função exponencial!

Algoritmo Tentativa e Erro - Análise

$T'(n) = \Omega(c_3^{\,n})$

T'(n) é limitado inferiormente por uma função exponencial!

T(n) também é limitado inferiormente por uma função exponencial!

Mesmo sem saber o limite superior, já temos informação suficiente para comparar o desempenho da versão tentativa e erro (exponencial ou pior) com a versão gulosa (linear).

Comparação: guloso vs. tentativa e erro

Guloso:

- escolha ótima local na expectativa de uma solução ótima global
- não reconsidera as escolhas já feitas (não há backtracking)
- solução ótima global não é garantida
- mas mesmo uma solução não-ótima pode ser uma boa aproximação
- mais eficientes

Tentativa e erro:

- reconsidera as escolhas já feitas, ou seja, há backtracking
- garante solução ótima global
- complexidade exponencial