Problemas clássicos de Programação Dinâmica

Tiago Montalvão

UFRJ

Competições de Algoritmos e Programação UFRJ

SEMANA DA COMPUTAÇÃO UFRJ

Motivação

Seja:

$$X = 1+1+1+1+1+1+1+1+1+1+1+1$$

Qual o valor de X?

Motivação

Seja:

$$X = 1+1+1+1+1+1+1+1+1+1+1+1$$

Qual o valor de X? 12

Motivação

Seja:

$$X = 1+1+1+1+1+1+1+1+1+1+1+1+1$$

Qual o valor de X?

Motivação

Seja:

$$X = 1+1+1+1+1+1+1+1+1+1+1+1+1$$

Qual o valor de X? 13

Motivação

Por que a primeira pergunta demorou mais que a segunda a ser respondida?

Porque não tínhamos nenhuma informação prévia quando respondemos à primeira pergunta.

Motivação

Por que a primeira pergunta demorou mais que a segunda a ser respondida?

Porque não tínhamos nenhuma informação prévia quando respondemos à primeira pergunta.

Já para a segunda resposta, tínhamos calculado antes a soma de todos os termos, exceto o último, que foi feito de forma imediata.

Motivação

Por que a primeira pergunta demorou mais que a segunda a ser respondida?

Porque não tínhamos nenhuma informação prévia quando respondemos à primeira pergunta.

Já para a segunda resposta, tínhamos calculado antes a soma de todos os termos, exceto o último, que foi feito de forma imediata.

Isto é programação dinâmica!!

Programação dinâmica

Programação Dinâmica:

Técnica de programação utilizada para resolver um problema complexo, quebrando-o em uma coleção de subproblemas, resolvendo cada um destes subproblemas apenas uma vez e guardando suas soluções.

Programação dinâmica

Programação Dinâmica:

1. Reconhecer os subproblemas
2. Resolver os subproblemas, geralmente através de uma relação de recorrência
3. Combinar as soluções para resolver problema original
4. Armazenar solução

Fibonacci

Escreva um algoritmo que calcule o n-ésimo termo da sequência de Fibonacci.

Fibonacci

Escreva um algoritmo que calcule o n-ésimo termo da sequência de Fibonacci.

A sequência de Fibonacci é definida por:

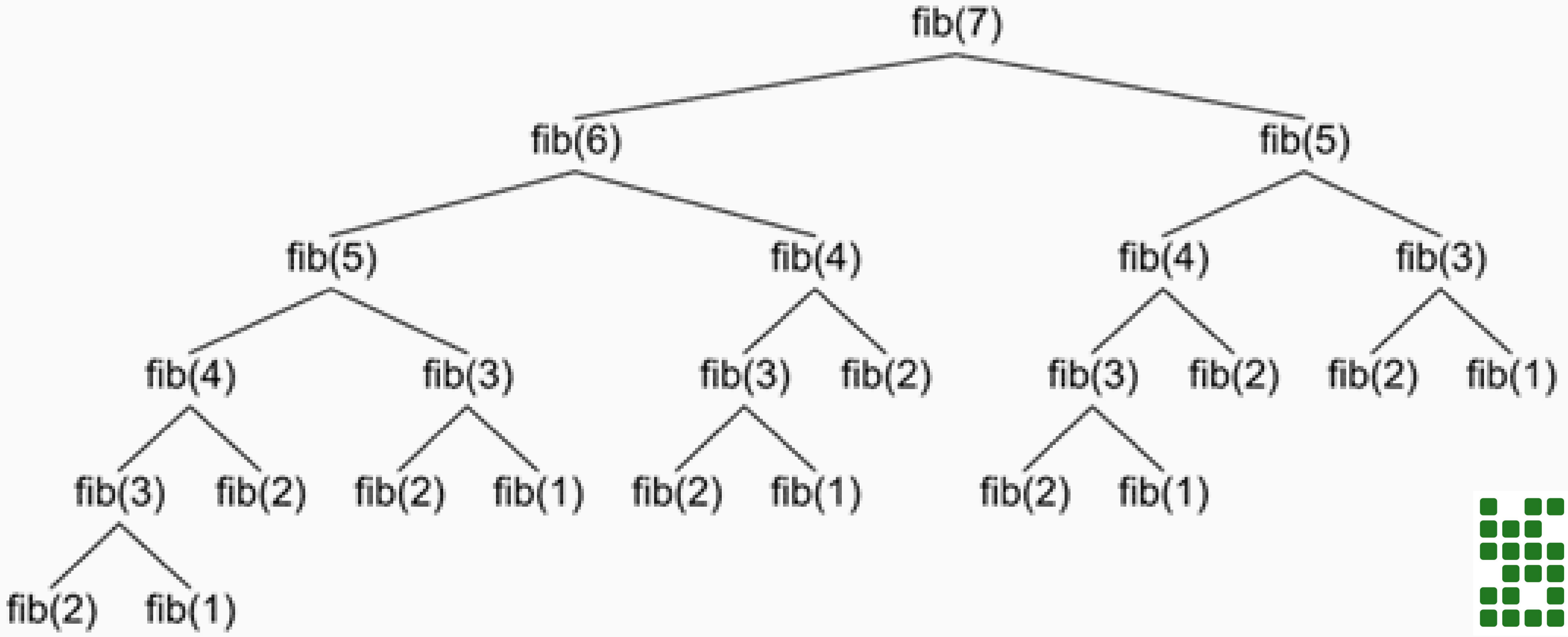
- $F_1 = 1$
- $F_2 = 1$
- $F_n = F_{n-1} + F_{n-2}$, para $n > 2$

Fibonacci

Escreva um algoritmo que calcule o n-ésimo termo da sequência de Fibonacci.

```
fibonacci(n):
    se n = 1 ou n = 2: retorna 1
    senão: retorna fibonacci(n-1) + fibonacci(n-2)
```

Fibonacci


fib(7)
fib(6)
fib(5)
fib(5)
fib(4)
fib(4)
fib(3)
fib(4)
fib(3)
fib(3)
fib(2)
fib(3)
fib(2)
fib(2)
fib(1)
fib(3)
fib(2)
fib(2)
fib(1)
fib(2)
fib(1)
fib(2)
fib(1)
fib(2)
fib(1)

Fibonacci

Como vimos, calcular cegamente os valores da função é muito ruim. Em termos de complexidade de tempo, esta solução é $O(2^n)$.

Fibonacci

Como vimos, calcular cegamente os valores da função é muito ruim. Em termos de complexidade de tempo, esta solução é $O(2^n)$.

Programação dinâmica resolve este problema, ao armazenar os valores já calculados e não calculá-los novamente.

Fibonacci

Como vimos, calcular cegamente os valores da função é muito ruim. Em termos de complexidade de tempo, esta solução é $O(2^n)$.

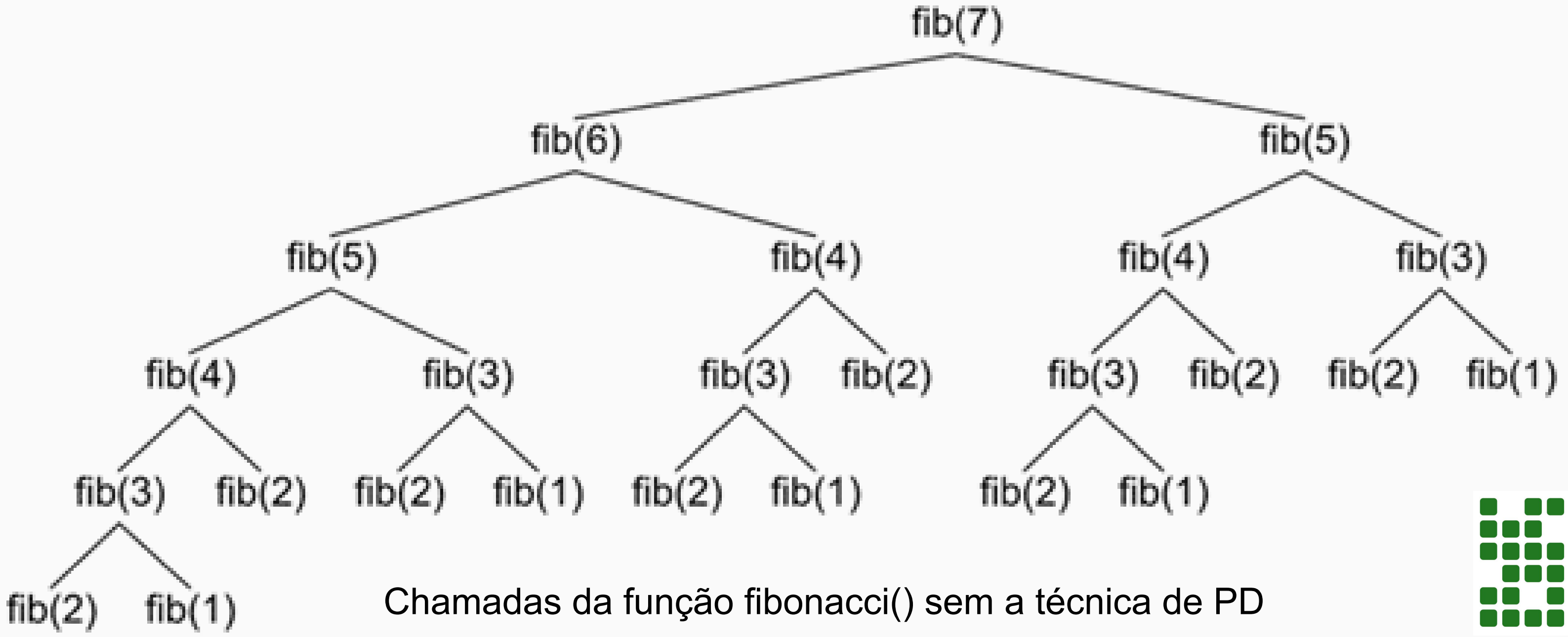
Programação dinâmica resolve este problema, ao armazenar os valores já calculados e não calculá-los novamente.

Vejamos como modificar o algoritmo escrito anteriormente para levar em consideração esta modificação:

Fibonacci

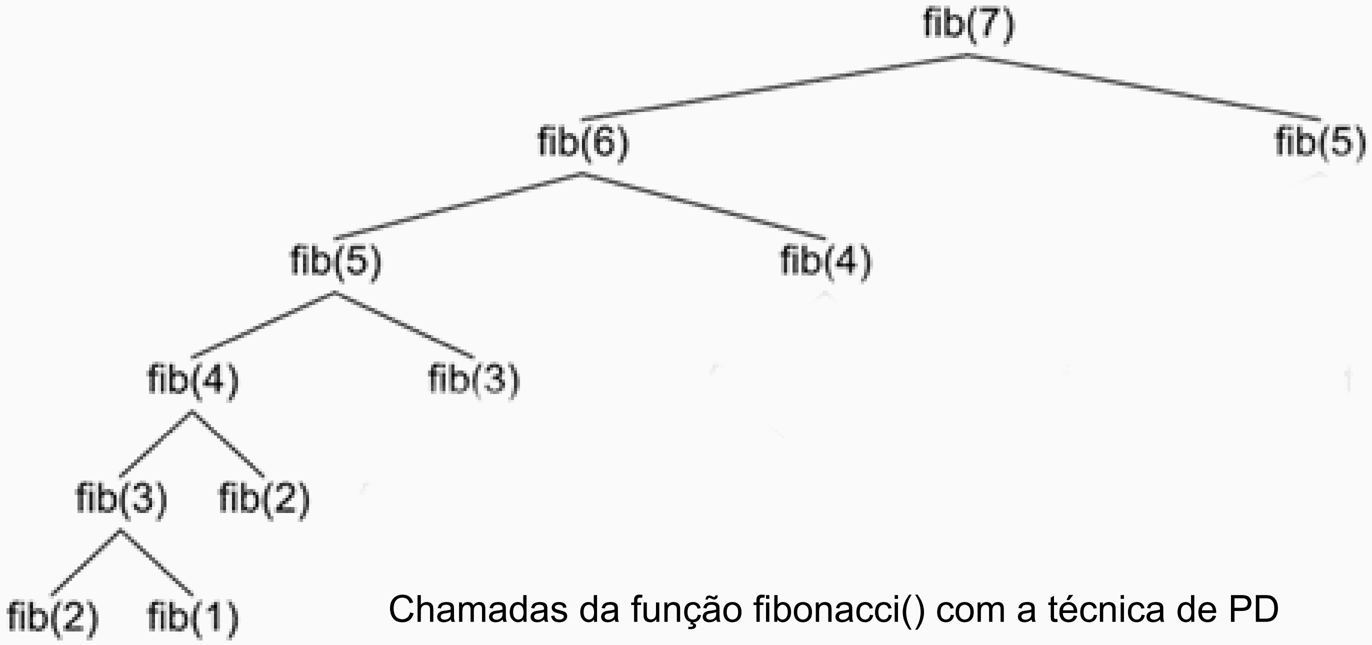
```
fibonacci(n):
    se fibonacci(n) já foi calculado: retorna valorCalculado(n)
    senão se n = 1 ou n = 2: retorna 1
    senão:  calcula fibonacci(n-1) + fibonacci(n-2)
            armazena resultado em valorCalculado(n)
            retorna valorCalculado(n)
```

Fibonacci



Chamadas da função fibonacci() sem a técnica de PD

Fibonacci



Chamadas da função fibonacci() com a técnica de PD

Implementação da solução

A implementação fornecida para calcular o n-ésimo termo da sequência de Fibonacci é chamada de **top-down**.

Implementação da solução

A implementação fornecida para calcular o n-ésimo termo da sequência de Fibonacci é chamada de **top-down**.

A principal característica de tal solução é o caráter recursivo.

Implementação da solução

A implementação fornecida para calcular o n-ésimo termo da sequência de Fibonacci é chamada de **top-down**.

A principal característica de tal solução é o caráter recursivo.

Para calcular o valor de F_7, não tínhamos calculados os valores menores, que foram sendo calculados quando necessário e armazenados.

Implementação da solução

A implementação fornecida para calcular o n-ésimo termo da sequência de Fibonacci é chamada de **top-down**.

A principal característica de tal solução é o caráter recursivo.

Para calcular o valor de F_7, não tínhamos calculados os valores menores, que foram sendo calculados quando necessário e armazenados.

Tal técnica de armazenamento e checagem se um problema foi calculado ou não é chamado **memoização**.

Implementação da solução

Uma outra alternativa, que será focada a partir de agora na apresentação é a implementação **bottom-up**.

Implementação da solução

Uma outra alternativa, que será focada a partir de agora na apresentação é a implementação **bottom-up**.

Rigorosamente falando, programação dinâmica é implementada apenas utilizando este método.

Implementação da solução

Uma outra alternativa, que será focada a partir de agora na apresentação é a implementação **bottom-up**.

Rigorosamente falando, programação dinâmica é implementada apenas utilizando este método.

Em uma implementação bottom-up, os valores são calculados em determinada ordem, tal que para calcular um valor maior, todos os menores com certeza já foram calculados.

Implementação da solução

Uma outra alternativa, que será focada a partir de agora na apresentação é a implementação **bottom-up**.

Rigorosamente falando, programação dinâmica é implementada apenas utilizando este método.

Em uma implementação bottom-up, os valores são calculados em determinada ordem, tal que para calcular um valor maior, todos os menores com certeza já foram calculados. Vejamos a implementação bottom-up de Fibonacci:

Fibonacci

```
fibonacci(n):
    fibo(1) = 1
    fibo(2) = 1
    para i ← 3 até n:
        fibo(i) ← fibo(i-1) + fibo(i-2)
```

Implementação da solução

A escolha de qual método utilizar fica ao gosto de quem irá implementar a solução. Algumas observações apenas são feitas:

Implementação da solução

A escolha de qual método utilizar fica ao gosto de quem irá implementar a solução. Algumas observações apenas são feitas:

Bottom-Up	Top-Down

Implementação da solução

A escolha de qual método utilizar fica ao gosto de quem irá implementar a solução. Algumas observações apenas são feitas:

Bottom-Up	Top-Down
Atribuir todos os casos base da recorrência antes de começar a iteração principal	

Implementação da solução

A escolha de qual método utilizar fica ao gosto de quem irá implementar a solução. Algumas observações apenas são feitas:

Bottom-Up	Top-Down
Atribuir todos os casos base da recorrência antes de começar a iteração principal	Definir os casos base da recorrência na própria função recursiva

Implementação da solução

A escolha de qual método utilizar fica ao gosto de quem irá implementar a solução. Algumas observações apenas são feitas:

Bottom-Up	Top-Down
Atribuir todos os casos base da recorrência antes de começar a iteração principal	Definir os casos base da recorrência na própria função recursiva
Garantir que todos os subproblemas foram calculados quando for calcular um subproblema	

Implementação da solução

A escolha de qual método utilizar fica ao gosto de quem irá implementar a solução. Algumas observações apenas são feitas:

Bottom-Up	Top-Down
Atribuir todos os casos base da recorrência antes de começar a iteração principal	Definir os casos base da recorrência na própria função recursiva
Garantir que todos os subproblemas foram calculados quando for calcular um subproblema	Garantir que todas as soluções calculadas estão sendo armazenadas para não haver cálculo redundante

Programação dinâmica

Muitos problemas que são resolvidos com a técnica de programação dinâmica se enquadram como problemas de otimização combinatória.

Programação dinâmica

Muitos problemas que são resolvidos com a técnica de programação dinâmica se enquadram como problemas de otimização combinatória.

Problema de otimização combinatória:

Problema de otimização em um conjunto finito, dado por uma função objetivo a ser minimizada ou maximizada, respeitando uma série de restrições.

Programação dinâmica

Muitos problemas que são resolvidos com a técnica de programação dinâmica se enquadram como problemas de otimização combinatória.

Problema de otimização combinatória:

Problema de otimização em um conjunto finito, dado por uma função objetivo a ser minimizada ou maximizada, respeitando uma série de restrições.

Estes conceitos ficarão mais claros quando aplicados a exemplos.

Subsequência contígua de maior soma

Descrição:

Dada uma sequência de números, ache a maior soma a ser formada utilizando 1 ou mais elementos adjacentes na sequência.

Subsequência contígua de maior soma

Descrição:

Dada uma sequência de números, ache a maior soma a ser formada utilizando 1 ou mais elementos adjacentes na sequência.

Exemplo:

10	5	-17	20	50	-1	3	-30	10

Subsequência contígua de maior soma

Descrição:

Dada uma sequência de números, ache a maior soma a ser formada utilizando 1 ou mais elementos adjacentes na sequência.

Exemplo:

10	5	-17	20	50	-1	3	-30	10

Subsequência contígua de maior soma

Ideias?

10	5	-17	20	50	-1	3	-30	10

Ideia #1

Ideia #1:

Para cada subsequência contígua:

some todos os elementos da subsequência

armazene o maior valor calculado

Ideia #1

```
maiorSoma ← -∞
para i ← 1 até n:
    para j ← 1 até n:
        somaAtual ← soma(i,j)
        if (somaAtual > maiorSoma):
            maiorSoma ← somaAtual
```

Ideia #1

```
maiorSoma ← -∞
para i ← 1 até n:                              O(n)
    para j ← 1 até n:
        somaAtual ← soma(i,j)
        if (somaAtual > maiorSoma):
            maiorSoma ← somaAtual
```

Ideia #1

```
maiorSoma ← -∞

para i ← 1 até n:                           O(n)

    para j ← 1 até n:                       O(n)

        somaAtual ← soma(i,j)

        if (somaAtual > maiorSoma):

            maiorSoma ← somaAtual
```

Ideia #1

```
maiorSoma ← -∞

para i ← 1 até n:                                O(n)

    para j ← 1 até n:                            O(n)

        somaAtual ← soma(i,j)                    O(n)

        if (somaAtual > maiorSoma):

            maiorSoma ← somaAtual
```

Ideia #1

Complexidade de tempo:

Ideia #1

Complexidade de tempo:

$$O(n*n*n) = O(n^3)$$

Ideia #1

Complexidade de tempo:

$$O(n*n*n) = O(n^3)$$

Podemos fazer melhor??

Ideia #1

Complexidade de tempo:

$$O(n*n*n) = O(n^3)$$

Podemos fazer melhor?? Sim!

Ideia #2

Ideia #2:

Armazenar somas acumuladas do início da sequência até cada elemento

Para cada subsequência contígua:

ache a soma da subsequência utilizando a informação pré-processada

armazene o maior valor calculado

Ideia #2

A diferença é basicamente na forma de calcular a soma de cada subsequência.

Com o uso de um vetor auxiliar *somaAcumulada* podemos atingir este objetivo e calcular em tempo constante esta soma.

O vetor é definido como:

- somaAcumulada(0) = 0
- somaAcumulada(n) = sequencia(n) + somaAcumulada(n-1)

Ideia #2

10	5	-17	20	50	-1	3	-30	10

Sequência original

Ideia #2

10	5	-17	20	50	-1	3	-30	10

Sequência original

10	15	-2	18	68	67	70	40	50

Sequência formada pelas somas acumuladas

Ideia #2

Para calcular a soma entre as posições i e j, agora basta calcular somaAcumulada(j) - somaAcumulada(i-1).

Ideia #2

Para calcular a soma entre as posições i e j, agora basta calcular somaAcumulada(j) - somaAcumulada(i-1).

Exemplo: calcular soma entre as posições 8 e 3.

10	5	-17	20	50	-1	3	-30	10

Ideia #2

Exemplo: calcular soma entre as posições 8 e 3.

10	5	-17	20	50	-1	3	-30	10

Resposta =

Ideia #2

Exemplo: calcular soma entre as posições 8 e 3.

10	5	-17	20	50	-1	3	-30	10

Resposta = 40

Ideia #2

Exemplo: calcular soma entre as posições 8 e 3.

10	5	-17	20	50	-1	3	-30	10

10	5	-17	20	50	-1	3	-30	10

Resposta = 40 - 15

Ideia #2

Exemplo: calcular soma entre as posições 8 e 3.

10	5	-17	20	50	-1	3	-30	10

10	5	-17	20	50	-1	3	-30	10

10	5	-17	20	50	-1	3	-30	10

Resposta = 40 - 15 = 25

Ideia #2

```
calcula vetor somaAcumulada

maiorSoma ← -∞

para i ← 1 até n:                                              O(n)

    para j ← 1 até n:                                          O(n)

        somaAtual ← somaAcumulada(j) - somaAcumulada(i-1)      O(1)

        if (somaAtual > maiorSoma):

            maiorSoma ← somaAtual
```

Ideia #2

Complexidade de tempo:

Ideia #2

Complexidade de tempo:

$$O(n*n) = O(n^2)$$

Ideia #2

Complexidade de tempo:

$O(n*n) = O(n^2)$

Podemos fazer melhor??

Ideia #2

Complexidade de tempo:

$$O(n*n) = O(n^2)$$

Podemos fazer melhor?? Sim!

Ideia #3

Programação dinâmica!

Ideia #3

Programação dinâmica!

Manter um vetor auxiliar *pd*, tal que pd(i) armazena a maior soma de uma subsequência contígua que termina na posição i, utilizando a posição i.

Ideia #3

Programação dinâmica!

Manter um vetor auxiliar *pd*, tal que pd(i) armazena a maior soma de uma subsequência contígua que termina na posição i, utilizando a posição i.

Para o nosso exemplo:

Ideia #3

10	5	-17	20	50	-1	3	-30	10

Sequência original

Ideia #3

10	5	-17	20	50	-1	3	-30	10

Sequência original

10	15	-2	20	70	69	72	42	52

Valor da maior subsequência contígua terminando em cada posição

Ideia #3

Com estas informações, podemos simplesmente achar o maior valor neste vetor *pd* e esta será a resposta.

Ideia #3

Com estas informações, podemos simplesmente achar o maior valor neste vetor *pd* e esta será a resposta. Por quê?

Ideia #3

Com estas informações, podemos simplesmente achar o maior valor neste vetor *pd* e esta será a resposta. Por quê?

A subsequência contígua de maior soma termina necessariamente em algum elemento da sequência original. Como estamos olhando cada término possível, estamos verificando todas os possíveis resultados.

Ideia #3

Com estas informações, podemos simplesmente achar o maior valor neste vetor *pd* e esta será a resposta. Por quê?

A subsequência contígua de maior soma termina necessariamente em algum elemento da sequência original. Como estamos olhando cada término possível, estamos verificando todas os possíveis resultados.

Mas como montar o vetor *pd*?

Ideia #3

Lembre-se dos passos a serem executados em um problema de PD:

1. Reconhecer os subproblemas
2. Resolver os subproblemas, geralmente através de uma relação de recorrência
3. Combinar as soluções para resolver problema original
4. Armazenar solução

Ideia #3

Podemos considerar que estamos querendo calcular o valor pd(i), com pd(j) já calculado, $\forall$ j < i. Sendo assim, temos duas possibilidades para a maior soma terminando na posição i $\geq$ 1:

Ideia #3

Podemos considerar que estamos querendo calcular o valor $pd(i)$, com $pd(j)$ já calculado, $\forall j < i$. Sendo assim, temos duas possibilidades para a maior soma terminando na posição $i \geq 1$:

1. $pd(i)$ terá o valor apenas do elemento na posição i

Ideia #3

Podemos considerar que estamos querendo calcular o valor pd(i), com pd(j) já calculado, $\forall$ j < i. Sendo assim, temos duas possibilidades para a maior soma terminando na posição i ≥ 1:

1. pd(i) terá o valor apenas do elemento na posição i
2. pd(i) será o valor do elemento na posição i somado com a maior soma até a posição i-1

Ideia #3

Podemos considerar que estamos querendo calcular o valor pd(i), com pd(j) já calculado, $\forall$ j < i. Sendo assim, temos duas possibilidades para a maior soma terminando na posição i ≥ 1:

1. pd(i) = sequencia(i)
2. pd(i) será o valor do elemento na posição i somado com a maior soma até a posição i-1

Ideia #3

Podemos considerar que estamos querendo calcular o valor pd(i), com pd(j) já calculado, $\forall$ j < i. Sendo assim, temos duas possibilidades para a maior soma terminando na posição i ≥ 1:

1. pd(i) = sequencia(i)
2. pd(i) = sequencia(i) + pd(i-1)

Ideia #3

Podemos considerar que estamos querendo calcular o valor pd(i), com pd(j) já calculado, $\forall$ j < i. Sendo assim, temos duas possibilidades para a maior soma terminando na posição i ≥ 1:

1. pd(i) = sequencia(i)
2. pd(i) = sequencia(i) + pd(i-1)

Como queremos sempre a maior soma, para cada índice fazemos:

Ideia #3

Podemos considerar que estamos querendo calcular o valor pd(i), com pd(j) já calculado, $\forall$ j < i. Sendo assim, temos duas possibilidades para a maior soma terminando na posição i ≥ 1:

1. pd(i) = sequencia(i)
2. pd(i) = sequencia(i) + pd(i-1)

Como queremos sempre a maior soma, para cada índice fazemos:

pd(i) = max(sequencia(i), sequencia(i) + pd(i-1))

Ideia #3

10	5	-17	20	50	-1	3	-30	10

Sequência original

10	15	-2	20	70	69	72	42	52

Valor da maior subsequência contígua terminando em cada posição

Ideia #3

```
pd(0) ← 0
maiorSoma ← -∞
para i ← 1 até n:                                          O(n)
    pd(i) ← max(sequencia(i), sequencia(i) + pd(i-1))      O(1)
    if (pd(i) > maiorSoma):
        maiorSoma ← pd(i)
```

Ideia #3

Complexidade de tempo:

Ideia #3

Complexidade de tempo:

$O(n)$

Ideia #3

Complexidade de tempo:

$O(n)$

Podemos fazer melhor??

Ideia #3

Complexidade de tempo:

$O(n)$

Podemos fazer melhor?? Não!

Soma máxima de subsequência com elementos não adjacentes

Descrição:

Dada uma sequência de números, ache a maior soma a ser formada utilizando elementos não adjacentes na sequência.

Soma máxima de subsequência com elementos não adjacentes

Descrição:

Dada uma sequência de números, ache a maior soma a ser formada utilizando elementos não adjacentes na sequência.

Exemplo:

10	5	-17	20	50	-1	3	-30	10

Soma máxima de subsequência com elementos não adjacentes

Descrição:

Dada uma sequência de números, ache a maior soma a ser formada utilizando elementos não adjacentes na sequência.

Exemplo:

10	5	-17	20	50	-1	3	-30	10

Soma máxima de subsequência com elementos não adjacentes

Ideias?

10	5	-17	20	50	-1	3	-30	10

Ideia #1

Ideia #1:

Para cada subsequência:

se tiver elementos adjacentes ignora e continua

some todos os elementos da subsequência

armazene o maior valor calculado

Ideia #1

O problema imediato desta solução é que gerar todas as subsequências de uma sequência leva tempo proporcional à quantidade de subsequências da sequência original.

Ideia #1

O problema imediato desta solução é que gerar todas as subsequências de uma sequência leva tempo proporcional à quantidade de subsequências da sequência original.

Esta quantidade é de 2^n.

Ideia #1

O problema imediato desta solução é que gerar todas as subsequências de uma sequência leva tempo proporcional à quantidade de subsequências da sequência original.

Esta quantidade é de 2^n.

Portanto, a complexidade deste algoritmo proposto é $O(2^n)$.

Ideia #1

O problema imediato desta solução é que gerar todas as subsequências de uma sequência leva tempo proporcional à quantidade de subsequências da sequência original.

Esta quantidade é de 2^n.

Portanto, a complexidade deste algoritmo proposto é $O(2^n)$.

Podemos fazer melhor??

Ideia #1

O problema imediato desta solução é que gerar todas as subsequências de uma sequência leva tempo proporcional à quantidade de subsequências da sequência original.

Esta quantidade é de 2^n.

Portanto, a complexidade deste algoritmo proposto é $O(2^n)$.

Podemos fazer melhor?? Sim!

Ideia #2

Programação dinâmica!

Ideia #2

Programação dinâmica!

Manter um vetor auxiliar *pd*, tal que pd(i) armazena a maior soma de uma subsequência sem elementos adjacentes, que termina na posição i, utilizando ou não a posição i.

Ideia #2

Programação dinâmica!

Manter um vetor auxiliar *pd*, tal que pd(i) armazena a maior soma de uma subsequência sem elementos adjacentes, que termina na posição i, utilizando ou não a posição i.

Ou seja, é a resposta para o subproblema da sequência de i elementos começando no 1.

Ideia #2

Programação dinâmica!

Manter um vetor auxiliar *pd*, tal que pd(i) armazena a maior soma de uma subsequência sem elementos adjacentes, que termina na posição i, utilizando ou não a posição i.

Ou seja, é a resposta para o subproblema da sequência de i elementos começando no 1.

Para o nosso exemplo:

Ideia #2

10	5	-17	20	50	-1	3	-30	10

Sequência original

10	10	10	30	60	60	63	63	73

Valor da maior subsequência sem elementos adjacentes até cada posição

Ideia #2

Podemos considerar que estamos querendo calcular o valor $pd(i)$, com $pd(j)$ já calculado, $\forall\ j < i$. Sendo assim, temos duas possibilidades para a maior soma terminando na posição $i \geq 1$:

Ideia #2

Podemos considerar que estamos querendo calcular o valor pd(i), com pd(j) já calculado, $\forall$ j < i. Sendo assim, temos duas possibilidades para a maior soma terminando na posição i ≥ 1:

1. pd(i) não terá o elemento na posição i

Ideia #2

Podemos considerar que estamos querendo calcular o valor pd(i), com pd(j) já calculado, $\forall$ j < i. Sendo assim, temos duas possibilidades para a maior soma terminando na posição i ≥ 1:

1. pd(i) não terá o elemento na posição i
2. pd(i) terá o elemento na posição i

Ideia #2

Podemos considerar que estamos querendo calcular o valor pd(i), com pd(j) já calculado, $\forall$ j < i. Sendo assim, temos duas possibilidades para a maior soma terminando na posição i ≥ 1:

1. pd(i) = pd(i-1)
2. pd(i) terá o elemento na posição i

Ideia #2

Podemos considerar que estamos querendo calcular o valor pd(i), com pd(j) já calculado, ∀ j < i. Sendo assim, temos duas possibilidades para a maior soma terminando na posição i ≥ 1:

1. pd(i) = pd(i-1)
2. pd(i) = sequencia(i) + pd(i-2)

Ideia #2

Podemos considerar que estamos querendo calcular o valor pd(i), com pd(j) já calculado, $\forall$ j < i. Sendo assim, temos duas possibilidades para a maior soma terminando na posição i ≥ 1:

1. pd(i) = pd(i-1)
2. pd(i) = sequencia(i) + pd(i-2)

Como queremos sempre a maior soma, para cada índice fazemos:

Ideia #2

Podemos considerar que estamos querendo calcular o valor pd(i), com pd(j) já calculado, $\forall$ $j < i$. Sendo assim, temos duas possibilidades para a maior soma terminando na posição $i \geq 1$:

1. pd(i) = pd(i-1)
2. pd(i) = sequencia(i) + pd(i-2)

Como queremos sempre a maior soma, para cada índice fazemos:

pd(i) = max(pd(i-1), sequencia(i) + pd(i-2))

Ideia #2

Faltam apenas os casos base:

Ideia #2

Faltam apenas os casos base:

Como, para calcular pd(i), sempre olhamos para uma posição anterior e duas posições anteriores, o primeiro valor a ser calculado deve ter pelo menos dois já calculados previamente.

Ideia #2

Faltam apenas os casos base:

Como, para calcular pd(i), sempre olhamos para uma posição anterior e duas posições anteriores, o primeiro valor a ser calculado deve ter pelo menos dois já calculados previamente.

Portanto, definimos:

pd(0) = 0

pd(1) = sequencia(1)

Ideia #2

10	5	-17	20	50	-1	3	-30	10

Sequência original

10	10	10	30	60	60	63	63	73

Valor da maior subsequência sem elementos adjacentes até cada posição

Maior subsequência comum

Descrição:

Dadas duas strings s_1 e s_2 , de tamanhos n e m respectivamente, achar o tamanho da maior sequência que é subsequência de ambas s_1 e s_2.

Maior subsequência comum

Descrição:

Dadas duas strings s_1 e s_2 , de tamanhos n e m respectivamente, achar o tamanho da maior sequência que é subsequência de ambas s_1 e s_2.

Exemplo:

s_1 = GAC

s_2 = AGCAT

Maior subsequência comum

Descrição:

Dadas duas strings s_1 e s_2 , de tamanhos n e m respectivamente, achar o tamanho da maior sequência que é subsequência de ambas s_1 e s_2.

Exemplo:

s_1 = GAC

s_2 = AGCAT

Maior subsequência comum

Descrição:

Dadas duas strings s_1 e s_2 , de tamanhos n e m respectivamente, achar o tamanho da maior sequência que é subsequência de ambas s_1 e s_2.

Ideias?

Ideia #1

Uma solução é gerar todas as subsequências de cada uma das strings inciais e compará-las.

Ideia #1

Uma solução é gerar todas as subsequências de cada uma das strings inciais e compará-las.

A complexidade desta solução é $O(2^{n+m})$, pois serão 2^n subsequências da primeira string e 2^m subsequências da segunda string, sendo feita uma comparação para cada par.

Ideia #1

Uma solução é gerar todas as subsequências de cada uma das strings inciais e compará-las.

A complexidade desta solução é $O(2^{n+m})$, pois serão 2^n subsequências da primeira string e 2^m subsequências da segunda string, sendo feita uma comparação para cada par.

Podemos fazer melhor??

Ideia #1

Uma solução é gerar todas as subsequências de cada uma das strings inciais e compará-las.

A complexidade desta solução é $O(2^{n+m})$, pois serão 2^n subsequências da primeira string e 2^m subsequências da segunda string, sendo feita uma comparação para cada par.

Podemos fazer melhor?? Sim!

Ideia #2

Programação dinâmica!

Ideia #2

Programação dinâmica!

Manter uma matriz auxiliar *pd*, tal que pd(i)(j) é a resposta para o subproblema, considerando os prefixos de s_1 e s_2 de tamanhos i e j, respectivamente.

Ideia #2

Programação dinâmica!

Manter uma matriz auxiliar *pd*, tal que pd(i)(j) é a resposta para o subproblema, considerando os prefixos de s_1 e s_2 de tamanhos i e j, respectivamente.

Uma vez que esta matriz esteja preenchida, basta responder com o valor presente em pd(n)(m).

Ideia #2

Programação dinâmica!

Manter uma matriz auxiliar *pd*, tal que pd(i)(j) é a resposta para o subproblema, considerando os prefixos de s_1 e s_2 de tamanhos i e j, respectivamente.

Uma vez que esta matriz esteja preenchida, basta responder com o valor presente em pd(n)(m).

Mas como montar esta matriz?

Ideia #2

Um subproblema agora é definido por dois valores: i e j.

Ideia #2

Um subproblema agora é definido por dois valores: i e j.

Denote por s(1...i) o prefixo de tamanho i de s.

Ideia #2

Um subproblema agora é definido por dois valores: i e j.

Denote por s(1...i) o prefixo de tamanho i de s.

Se a última letra de $s_1(1...i)$ for igual a $s_2(1...j)$, podemos colocar esta letra na subsequência.

Ideia #2

Um subproblema agora é definido por dois valores: i e j.

Denote por s(1...i) o prefixo de tamanho i de s.

Se a última letra de s_1(1...i) for igual a s_2(1...j), podemos colocar esta letra na subsequência.

$s_1(1..3) = GAC$

$s_2(1..3) = AGC$

Ideia #2

Um subproblema agora é definido por dois valores: i e j.

Denote por s(1...i) o prefixo de tamanho i de s.

Se a última letra de $s_1(1...i)$ for igual a $s_2(1...j)$, podemos colocar esta letra na subsequência.

$s_1(1..3)$ = GAC

$s_2(1..3)$ = AGC

Ideia #2

Caso contrário, teremos um dos seguintes casos:

Ideia #2

Caso contrário, teremos um dos seguintes casos:

1. A última letra de s_1 estará na subsequência procurada; ou

Ideia #2

Caso contrário, teremos um dos seguintes casos:

1. A última letra de s_1 estará na subsequência procurada; ou
2. A última letra de s_2 estará na subsequência procurada; ou

Ideia #2

Caso contrário, teremos um dos seguintes casos:

1. A última letra de s_1 estará na subsequência procurada; ou
2. A última letra de s_2 estará na subsequência procurada; ou
3. Nem a última letra de s_1 nem a de s_2 estarão na subsequência procurada.

Ideia #2

Caso contrário, teremos um dos seguintes casos:

1. A última letra de s_1 estará na subsequência procurada; ou
2. A última letra de s_2 estará na subsequência procurada; ou
3. Nem a última letra de s_1 nem a de s_2 estarão na subsequência procurada.

$s_1(1..2)$ = GA

$s_2(1..5)$ = AGCAT

Ideia #2

Caso a última letra de $s_1(1...i)$ seja igual a $s_2(1...j)$, podemos colocar esta letra na subsequência.

Ideia #2

Caso a última letra de $s_1(1...i)$ seja igual a $s_2(1...j)$, podemos colocar esta letra na subsequência.

Caso contrário, precisamos testar o que acontece se colocarmos apenas uma delas.

Ideia #2

Caso a última letra de $s_1(1...i)$ seja igual a $s_2(1...j)$, podemos colocar esta letra na subsequência.

Caso contrário, precisamos testar o que acontece se colocarmos apenas uma delas.

Qual delas?

Ideia #2

Caso a última letra de $s_1(1...i)$ seja igual a $s_2(1...j)$, podemos colocar esta letra na subsequência.

Caso contrário, precisamos testar o que acontece se colocarmos apenas uma delas.

Qual delas? Não sabemos, por isso testamos todos os casos e vemos o melhor.

Ideia #2

Sendo assim, temos que:

$$
pd(i)(j) = \begin{cases} 0\text{, se } i = 0 \text{ ou } j = 0 \\ pd(i-1)(j-1) + 1\text{, se } s1(1...i).last = s2(1...j).last \\ \max(pd(i-1)(j), pd(i)(j-1))\text{, caso contrário} \end{cases}
$$

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0				
A/1				
G/2				
C/3				
A/4				
T/5				

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0			
G/2	0			
C/3	0			
A/4	0			
T/5	0			

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0			
G/2	0			
C/3	0			
A/4	0			
T/5	0			

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0			
G/2	0			
C/3	0			
A/4	0			
T/5	0			

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0	0		
G/2	0			
C/3	0			
A/4	0			
T/5	0			

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0	0		
G/2	0			
C/3	0			
A/4	0			
T/5	0			

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0	0		
G/2	0			
C/3	0			
A/4	0			
T/5	0			

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0	0		
G/2	0			
C/3	0			
A/4	0			
T/5	0			

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0	0	1	
G/2	0			
C/3	0			
A/4	0			
T/5	0			

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0	0	1	
G/2	0			
C/3	0			
A/4	0			
T/5	0			

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0	0	1	
G/2	0			
C/3	0			
A/4	0			
T/5	0			

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0	0	1	
G/2	0			
C/3	0			
A/4	0			
T/5	0			

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0	0	1	1
G/2	0			
C/3	0			
A/4	0			
T/5	0			

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0	0	1	1
G/2	0			
C/3	0			
A/4	0			
T/5	0			

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0	0	1	1
G/2	0	1		
C/3	0			
A/4	0			
T/5	0			

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0	0	1	1
G/2	0	1	1	
C/3	0			
A/4	0			
T/5	0			

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0	0	1	1
G/2	0	1	1	1
C/3	0			
A/4	0			
T/5	0			

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0	0	1	1
G/2	0	1	1	1
C/3	0	1		
A/4	0			
T/5	0			

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0	0	1	1
G/2	0	1	1	1
C/3	0	1	1	
A/4	0			
T/5	0			

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0	0	1	1
G/2	0	1	1	1
C/3	0	1	1	2
A/4	0			
T/5	0			

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0	0	1	1
G/2	0	1	1	1
C/3	0	1	1	2
A/4	0	1		
T/5	0			

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0	0	1	1
G/2	0	1	1	1
C/3	0	1	1	2
A/4	0	1	2	
T/5	0			

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0	0	1	1
G/2	0	1	1	1
C/3	0	1	1	2
A/4	0	1	2	2
T/5	0			

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0	0	1	1
G/2	0	1	1	1
C/3	0	1	1	2
A/4	0	1	2	2
T/5	0	1		

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0	0	1	1
G/2	0	1	1	1
C/3	0	1	1	2
A/4	0	1	2	2
T/5	0	1	2	

Ideia #2

	Ø/0	G/1	A/2	C/3
Ø/0	0	0	0	0
A/1	0	0	1	1
G/2	0	1	1	1
C/3	0	1	1	2
A/4	0	1	2	2
T/5	0	1	2	2

Troco de moeda

Descrição:

Dadas uma quantia fixa V de dinheiro a ser dada de troco e um conjunto finito de moedas C de valores distintos, determine a menor quantidade necessária de moedas para formar o troco.

Exemplo:

$C = \{1, 5, 10, 25, 50, 100\}$ $V = 34$

Troco de moeda

Descrição:

Dadas uma quantia fixa V de dinheiro a ser dada de troco e um conjunto finito de moedas C de valores distintos, determine a menor quantidade necessária de moedas para formar o troco.

Exemplo:

$C = \{1, 5, 10, 25, 50, 100\}$ $V = 34 = 25 + 5 + 1 + 1 + 1 + 1$

Troco de moeda

Descrição:

Dadas uma quantia fixa V de dinheiro a ser dada de troco e um conjunto finito de moedas C de valores distintos, determine a menor quantidade necessária de moedas para formar o troco.

Exemplo:

C = {1, 5, 10, 25, 50, 100}   V = 34   R: 6 moedas

Troco de moeda

Descrição:

Dadas uma quantia fixa V de dinheiro a ser dada de troco e um conjunto finito de moedas C de valores distintos, determine a menor quantidade necessária de moedas para formar o troco.

Como resolver para um conjunto C qualquer e uma quantia V qualquer?

Ideia #1

A primeira ideia que temos é sempre pegar a maior moeda possível no conjunto a cada ponto.

Ideia #1

A primeira ideia que temos é sempre pegar a maior moeda possível no conjunto a cada ponto. Um algoritmo para tal ideia seria o seguinte:

Ideia #1

A primeira ideia que temos é sempre pegar a maior moeda possível no conjunto a cada ponto. Um algoritmo para tal ideia seria o seguinte:

```
resposta ← 0
enquanto V > 0:
    resposta ← resposta + 1
    m ← maior moeda menor ou igual a V
    V ← V - m
```

Ideia #1

Este algoritmo tem a vantagem de ser bem simples de entender e de implementar.

Ideia #1

Este algoritmo tem a vantagem de ser bem simples de entender e de implementar. Porém está errado!

Ideia #1

Este algoritmo tem a vantagem de ser bem simples de entender e de implementar. Porém está errado!

Ele funciona para casos bem específicos da escolha dos valores das moedas do conjunto C. Vejamos outro exemplo:

Ideia #1

Este algoritmo tem a vantagem de ser bem simples de entender e de implementar. Porém está errado!

Ele funciona para casos bem específicos da escolha dos valores das moedas do conjunto C. Vejamos outro exemplo:

Exemplo:

$C = \{1, 5, 10, 17, 25, 50, 100\}$ $V = 34$

Troco de moeda

Este algoritmo tem a vantagem de ser bem simples de entender e de implementar. Porém está errado!

Ele funciona para casos bem específicos da escolha dos valores das moedas do conjunto C. Vejamos outro exemplo:

Exemplo:

$C = \{1, 5, 10, 17, 25, 50, 100\}$ $V = 34 = 25 + 5 + 1 + 1 + 1 + 1$

Troco de moeda

Este algoritmo tem a vantagem de ser bem simples de entender e de implementar. Porém está errado!

Ele funciona para casos bem específicos da escolha dos valores das moedas do conjunto C. Vejamos outro exemplo:

Exemplo:

C = {1, 5, 10, 17, 25, 50, 100}   V = 34 ~~= 25 + 5 + 1 + 1 + 1 + 1~~

Troco de moeda

Este algoritmo tem a vantagem de ser bem simples de entender e de implementar. Porém está errado!

Ele funciona para casos bem específicos da escolha dos valores das moedas do conjunto C. Vejamos outro exemplo:

Exemplo:

$C = \{1, 5, 10, 17, 25, 50, 100\}$ $\qquad V = 34 = 17 + 17$

Troco de moeda

Este algoritmo tem a vantagem de ser bem simples de entender e de implementar. Porém está errado!

Ele funciona para casos bem específicos da escolha dos valores das moedas do conjunto C. Vejamos outro exemplo:

Exemplo:

$C = \{1, 5, 10, 17, 25, 50, 100\}$ $V = 34$ R: 2 moedas

Troco de moeda

Precisamos de outro algoritmo que esteja correto para qualquer escolha de C e de V.

Troco de moeda

Precisamos de outro algoritmo que esteja correto para qualquer escolha de C e de V.

Aí entra a programação dinâmica.

Ideia #2

Ideia:

Manter um vetor *pd*, tal que pd(i) indica a quantidade mínima de moedas necessárias para se formar o valor i.

Ideia #2

Ideia:

Manter um vetor *pd*, tal que pd(i) indica a quantidade mínima de moedas necessárias para se formar o valor i.

Após o cálculo deste vetor, a resposta estará expressa em pd(V).

Ideia #2

Ideia:

Manter um vetor *pd*, tal que pd(i) indica a quantidade mínima de moedas necessárias para se formar o valor i.

Após o cálculo deste vetor, a resposta estará expressa em pd(V).

Mas como calcular este vetor?

Ideia #2

Suponha que queiramos calcular a quantidade mínima de moedas necessárias para se formar um valor i, já sabendo a resposta para todos os valores j < i.

Ideia #2

Suponha que queiramos calcular a quantidade mínima de moedas necessárias para se formar um valor i, já sabendo a resposta para todos os valores $j < i$.

Podemos neste instante usar qualquer moeda para formar a soma i, tal que seu valor não ultrapasse i.

Ideia #2

Suponha que queiramos calcular a quantidade mínima de moedas necessárias para se formar um valor i, já sabendo a resposta para todos os valores j < i.

Podemos neste instante usar qualquer moeda para formar a soma i, tal que seu valor não ultrapasse i.

Sendo assim, testamos todas as moedas que satisfazem esta condição e vemos a melhor opção.

Ideia #2

Exemplo:

$C = \{1, 5, 10, 17, 25, 50, 100\}$ $V = 34$

$34 =$

Ideia #2

Exemplo:

$C = \{1, 5, 10, 17, 25, 50, 100\}$ $V = 34$

$34 = 1 + 33$

Ideia #2

Exemplo:

$C = \{1, 5, 10, 17, 25, 50, 100\}$ $\quad$ $V = 34$

$34 = 1 + 33$

$ = 5 + 29$

Ideia #2

Exemplo:

$C = \{1, 5, 10, 17, 25, 50, 100\}$ $V = 34$

$34 = 1 + 33$

$= 5 + 29$

$= 10 + 24$

Ideia #2

Exemplo:

$C = \{1, 5, 10, 17, 25, 50, 100\}$ $V = 34$

$34 = 1 + 33$

$= 5 + 29$ $= 17 + 17$

$= 10 + 24$

Ideia #2

Exemplo:

C = {1, 5, 10, 17, 25, 50, 100} V = 34

34 = 1 + 33

= 5 + 29 = 17 + 17

= 10 + 24 = 25 + 9

Ideia #2

Exemplo:

C = {1, 5, 10, 17, 25, 50, 100} V = 34

34 = 1 + 33

= 5 + 29

= 10 + 24

= 17 + 17

= 25 + 9

= 50 - 16

Ideia #2

Exemplo:

C = {1, 5, 10, 17, 25, 50, 100} V = 34

34 = 1 + 33

= 5 + 29 = 17 + 17 = ~~50 - 16~~

= 10 + 24 = 25 + 9

Ideia #2

Exemplo:

$C = \{1, 5, 10, 17, 25, 50, 100\}$ $V = 34$

$34 = 1 + 33$

$= 5 + 29$ $= 17 + 17$ $= \cancel{50 - 16}$

$= 10 + 24$ $= 25 + 9$ $= 100 - 66$

Ideia #2

Exemplo:

$C = \{1, 5, 10, 17, 25, 50, 100\}$

$V = 34$

$34 = 1 + 33$

$= 5 + 29$

$= 10 + 24$

$= 17 + 17$

$= 25 + 9$

$= \cancel{50 - 16}$

$= \cancel{100 - 66}$

Ideia #2

Os slides anteriores foram a execução de uma iteração do algoritmo.

Ideia #2

Os slides anteriores foram a execução de uma iteração do algoritmo.

Como notado, uma vez que utilizada uma moeda, com valor menor ou igual ao valor atual, ficamos com um valor menor a ser formado.

Ideia #2

Os slides anteriores foram a execução de uma iteração do algoritmo.

Como notado, uma vez que utilizada uma moeda, com valor menor ou igual ao valor atual, ficamos com um valor menor a ser formado.

Mas este valor já foi calculado previamente e podemos simplesmente utilizar as soluções dos problemas menores para calcular o problema maior.

Ideia #2

Exemplo:

$C = \{1, 5, 10, 17, 25, 50, 100\}$ $V = 34$

$34 = 1 + 33$

$= 5 + 29$ $= 17 + 17$ $= \cancel{50 - 16}$

$= 10 + 24$ $= 25 + 9$ $= \cancel{100 - 66}$

Ideia #2

Exemplo:

$C = \{1, 5, 10, 17, 25, 50, 100\}$ $V = 34$

$34 = 1 + 33$

$= 5 + 29$ $= 17 + 17$

$= 10 + 24$ $= 25 + 9$

Ideia #2

Exemplo:

$C = \{1, 5, 10, 17, 25, 50, 100\}$ $\qquad V = 34$

$$pd(34) = \min \begin{cases} 1 + pd(33), & 1 + pd(17), \\ 1 + pd(29), & 1 + pd(9) \\ 1 + pd(24), & \end{cases}$$

Ideia #2

Sendo assim, podemos escrever a relação de recorrência para cada posição.

$$pd(i) = \min\{1 + pd(i - c(j)),\ \forall j \text{ tal que } c(j) \leq i\}$$

Ideia #2

Sendo assim, podemos escrever a relação de recorrência para cada posição.

$$pd(i) = \min\{1 + pd(i - c(j)),\ \forall j \text{ tal que } c(j) \leq i\},\ \text{ se } i \geq 1$$

Ideia #2

Sendo assim, podemos escrever a relação de recorrência para cada posição.

$pd(i) = \min\{1 + pd(i - c(j)),\ \forall j \text{ tal que } c(j) \leq i\},\ \text{ se } i \geq 1$

$pd(0) =$

Ideia #2

Sendo assim, podemos escrever a relação de recorrência para cada posição.

$pd(i) = \min\{1 + pd(i - c(j)),\ \forall j \text{ tal que } c(j) \leq i\},\ \text{ se } i \geq 1$

$pd(0) = 0$

Ideia #2

Sendo assim, podemos escrever a relação de recorrência para cada posição.

$$pd(i) = \min\{1 + pd(i - c(j)),\ \forall j \text{ tal que } c(j) \leq i\},\ \text{ se } i \geq 1$$

$$pd(0) = 0$$

Podemos portanto escrever o algoritmo que resolve o problema do troco de moedas.

Ideia #2

```
pd(0) ← 0
para i ← 1 até V:
    pd(i) ← ∞
    para j ← 1 até |C|:
        se c(j) ≤ i e pd(i - c(j)) + 1 < pd(i):
            pd(i) ← pd(i - c(j)) + 1
```

Programação dinâmica

Programação Dinâmica:

1. Reconhecer os subproblemas
2. Resolver os subproblemas, geralmente através de uma relação de recorrência
3. Combinar as soluções para resolver problema original
4. Armazenar solução

Programação dinâmica

Como visto, muitos problemas podem ser resolvidos com a técnica de programação dinâmica. Alguns dos vários problemas a serem estudados são:

- Problema da Mochila
- Subsequência crescente mais longa
- Distância de edição
- Caminhos mínimos entre todos os pares de vértices em um grafo
- etc...

Programação dinâmica

Para que a técnica seja bem compreendida e bem aplicada, é necessário bastante prática em problemas diversos.

Obrigado!

Problemas clássicos de Programação Dinâmica

Tiago Montalvão

UFRJ

Competições de Algoritmos e Programação UFRJ

SEMANA DA COMPUTAÇÃO UFRJ