

Projeto e Análise de Algoritmos

Programação dinâmica

Lehilton Pedrosa

Primeiro Semestre de 2020

Programação dinâmica

Vamos estudar problemas com algumas propriedades:

- ▶ **subestrutura ótima**

- ▶ decomponemos a instância em vários subproblemas
- ▶ a solução ótima corresponde a exatamente um deles
- ▶ normalmente, é uma generalização do problema original

- ▶ **sobreposição de subproblemas**

- ▶ uma instância é quebrada em várias instâncias menores
- ▶ a recursão recalcula uma mesma instância várias vezes

Vamos estudar problemas com algumas propriedades:

- ▶ **subestrutura ótima**

- ▶ decompomos a instância em vários **subproblemas**
- ▶ a solução ótima corresponde a exatamente um deles
- ▶ normalmente, é uma generalização do problema original

- ▶ **sobreposição de subproblemas**

- ▶ uma instância é quebrada em várias instâncias menores
- ▶ a recursão recalcula uma mesma instância **várias vezes**

Vamos estudar problemas com algumas propriedades:

- ▶ **subestrutura ótima**

- ▶ decompomos a instância em vários **subproblemas**
- ▶ a solução ótima corresponde a exatamente um deles
- ▶ normalmente, é uma generalização do problema original

- ▶ **sobreposição de subproblemas**

- ▶ uma instância é quebrada em várias instâncias menores
- ▶ a recursão recalcula uma mesma instância várias vezes

Vamos estudar problemas com algumas propriedades:

- ▶ **subestrutura ótima**

- ▶ decompomos a instância em vários **subproblemas**
- ▶ a solução ótima corresponde a exatamente um deles
- ▶ normalmente, é uma generalização do problema original

- ▶ **sobreposição de subproblemas**

- ▶ uma instância é quebrada em várias instâncias menores
- ▶ a recursão recalcula uma mesma instância várias vezes

Problemas e subproblemas

Vamos estudar problemas com algumas propriedades:

- ▶ **subestrutura ótima**

- ▶ decompomos a instância em vários **subproblemas**
- ▶ a solução ótima corresponde a exatamente um deles
- ▶ normalmente, é uma generalização do problema original

- ▶ **sobreposição de subproblemas**

- ▶ uma instância é quebrada em várias instâncias menores
- ▶ a recursão recalcula uma mesma instância várias vezes

Vamos estudar problemas com algumas propriedades:

- ▶ **subestrutura ótima**

- ▶ decompomos a instância em vários **subproblemas**
- ▶ a solução ótima corresponde a exatamente um deles
- ▶ normalmente, é uma generalização do problema original

- ▶ **sobreposição de subproblemas**

- ▶ uma instância é quebrada em várias instâncias menores
- ▶ a recursão recalcula uma mesma instância **várias vezes**

Vamos estudar problemas com algumas propriedades:

- ▶ **subestrutura ótima**

- ▶ decompomos a instância em vários **subproblemas**
- ▶ a solução ótima corresponde a exatamente um deles
- ▶ normalmente, é uma generalização do problema original

- ▶ **sobreposição de subproblemas**

- ▶ uma instância é quebrada em várias instâncias menores
- ▶ a recursão recalcula uma mesma instância **várias vezes**

Vamos estudar problemas com algumas propriedades:

- ▶ **subestrutura ótima**

- ▶ decompomos a instância em vários **subproblemas**
- ▶ a solução ótima corresponde a exatamente um deles
- ▶ normalmente, é uma generalização do problema original

- ▶ **sobreposição de subproblemas**

- ▶ uma instância é quebrada em várias instâncias menores
- ▶ a recursão recalcula uma mesma instância **várias vezes**

Programação dinâmica

Ideia:

- ▶ evitar o recálculo desnecessário de subproblemas
- ▶ guardar as soluções de subproblemas em uma **tabela**
- ▶ cada entrada é uma instância distinta do subproblema

Premissas da programação dinâmica:

- ▶ o número entradas da tabela é pequeno
- ▶ sabemos computar cada uma eficientemente

Programação dinâmica

Ideia:

- ▶ evitar o recálculo desnecessário de subproblemas
- ▶ guardar as soluções de subproblemas em uma **tabela**
- ▶ cada entrada é uma instância distinta do subproblema

Premissas da programação dinâmica:

- ▶ o número entradas da tabela é pequeno
- ▶ sabemos computar cada uma eficientemente

Programação dinâmica

Ideia:

- ▶ evitar o recálculo desnecessário de subproblemas
- ▶ guardar as soluções de subproblemas em uma **tabela**
- ▶ cada entrada é uma instância distinta do subproblema

Premissas da programação dinâmica:

- ▶ o número entradas da tabela é pequeno
- ▶ sabemos computar cada uma eficientemente

Programação dinâmica

Ideia:

- ▶ evitar o recálculo desnecessário de subproblemas
- ▶ guardar as soluções de subproblemas em uma **tabela**
- ▶ cada entrada é uma instância distinta do subproblema

Premissas da programação dinâmica:

- ▶ o número entradas da tabela é pequeno
- ▶ sabemos computar cada uma eficientemente

Programação dinâmica

Ideia:

- ▶ evitar o recálculo desnecessário de subproblemas
- ▶ guardar as soluções de subproblemas em uma **tabela**
- ▶ cada entrada é uma instância distinta do subproblema

Premissas da programação dinâmica:

- ▶ o número entradas da tabela é pequeno
- ▶ sabemos computar cada uma eficientemente

Programação dinâmica

Ideia:

- ▶ evitar o recálculo desnecessário de subproblemas
- ▶ guardar as soluções de subproblemas em uma **tabela**
- ▶ cada entrada é uma instância distinta do subproblema

Premissas da programação dinâmica:

- ▶ o número entradas da tabela é pequeno
- ▶ sabemos computar cada uma eficientemente

Programação dinâmica

Ideia:

- ▶ evitar o recálculo desnecessário de subproblemas
- ▶ guardar as soluções de subproblemas em uma **tabela**
- ▶ cada entrada é uma instância distinta do subproblema

Premissas da programação dinâmica:

- ▶ o número entradas da tabela é pequeno
- ▶ sabemos computar cada uma eficientemente

Problemas de otimização

Consideramos tipicamente **problemas de otimização**

- ▶ cada instância tem um conjunto de **soluções viáveis**
- ▶ cada uma delas tem um valor dado pela **função-objetivo**
- ▶ queremos alguma cujo valor é **mínimo ou máximo**

Uma solução viável com o melhor valor é chamada de **ótima**

Problemas de otimização

Consideramos tipicamente **problemas de otimização**

- ▶ cada instância tem um conjunto de **soluções viáveis**
- ▶ cada uma delas tem um valor dado pela **função-objetivo**
- ▶ queremos alguma cujo valor é **mínimo ou máximo**

Uma solução viável com o melhor valor é chamada de **ótima**

Problemas de otimização

Consideramos tipicamente **problemas de otimização**

- ▶ cada instância tem um conjunto de **soluções viáveis**
- ▶ cada uma delas tem um valor dado pela **função-objetivo**
- ▶ queremos alguma cujo valor é **mínimo ou máximo**

Uma solução viável com o melhor valor é chamada de **ótima**

Problemas de otimização

Consideramos tipicamente **problemas de otimização**

- ▶ cada instância tem um conjunto de **soluções viáveis**
- ▶ cada uma delas tem um valor dado pela **função-objetivo**
- ▶ queremos alguma cujo valor é **mínimo ou máximo**

Uma solução viável com o melhor valor é chamada de **ótima**

Problemas de otimização

Consideramos tipicamente **problemas de otimização**

- ▶ cada instância tem um conjunto de **soluções viáveis**
- ▶ cada uma delas tem um valor dado pela **função-objetivo**
- ▶ queremos alguma cujo valor é **mínimo ou máximo**

Uma solução viável com o melhor valor é chamada de **ótima**

Programação dinâmica

- ▶ Multiplicação de cadeias de matrizes

Parentização

Considere um produto de matrizes

$$M = M_1 \times M_2 \times \dots \times M_i \cdots \times M_n$$

- ▶ as dimensões das matrizes são dadas por um vetor b
- ▶ a matriz M_i tem b_{i-1} linhas e b_i colunas

Ordem de multiplicação:

- ▶ só podemos multiplicar matrizes aos pares
- ▶ devemos escolher uma **parentização** para o produto
- ▶ produto matrizes $m \times l$ e $l \times n$ faz $m \cdot l \cdot n$ multiplicações

Parentização

Considere um produto de matrizes

$$M = M_1 \times M_2 \times \dots \times M_i \cdots \times M_n$$

- ▶ as dimensões das matrizes são dadas por um vetor b
- ▶ a matriz M_i tem b_{i-1} linhas e b_i colunas

Ordem de multiplicação:

- ▶ só podemos multiplicar matrizes aos pares
- ▶ devemos escolher uma **parentização** para o produto
- ▶ produto matrizes $m \times l$ e $l \times n$ faz $m \cdot l \cdot n$ multiplicações

Parentização

Considere um produto de matrizes

$$M = M_1 \times M_2 \times \dots \times M_i \cdots \times M_n$$

- ▶ as dimensões das matrizes são dadas por um vetor b
- ▶ a matriz M_i tem b_{i-1} linhas e b_i colunas

Ordem de multiplicação:

- ▶ só podemos multiplicar matrizes aos pares
- ▶ devemos escolher uma **parentização** para o produto
- ▶ produto matrizes $m \times l$ e $l \times n$ faz $m \cdot l \cdot n$ multiplicações

Parentização

Considere um produto de matrizes

$$M = M_1 \times M_2 \times \dots \times M_i \cdots \times M_n$$

- ▶ as dimensões das matrizes são dadas por um vetor b
- ▶ a matriz M_i tem b_{i-1} linhas e b_i colunas

Ordem de multiplicação:

- ▶ só podemos multiplicar matrizes aos pares
- ▶ devemos escolher uma **parentização** para o produto
- ▶ produto matrizes $m \times l$ e $l \times n$ faz $m \cdot l \cdot n$ multiplicações

Parentização

Considere um produto de matrizes

$$M = M_1 \times M_2 \times \dots \times M_i \cdots \times M_n$$

- ▶ as dimensões das matrizes são dadas por um vetor b
- ▶ a matriz M_i tem b_{i-1} linhas e b_i colunas

Ordem de multiplicação:

- ▶ só podemos multiplicar matrizes aos pares
- ▶ devemos escolher uma **parentização** para o produto
- ▶ produto matrizes $m \times l$ e $l \times n$ faz $m \cdot l \cdot n$ multiplicações

Parentização

Considere um produto de matrizes

$$M = M_1 \times M_2 \times \dots \times M_i \cdots \times M_n$$

- ▶ as dimensões das matrizes são dadas por um vetor b
- ▶ a matriz M_i tem b_{i-1} linhas e b_i colunas

Ordem de multiplicação:

- ▶ só podemos multiplicar matrizes aos pares
- ▶ devemos escolher uma **parentização** para o produto
- ▶ produto matrizes $m \times l$ e $l \times n$ faz $m \cdot l \cdot n$ multiplicações

Parentização

Considere um produto de matrizes

$$M = M_1 \times M_2 \times \dots \times M_i \cdots \times M_n$$

- ▶ as dimensões das matrizes são dadas por um vetor b
- ▶ a matriz M_i tem b_{i-1} linhas e b_i colunas

Ordem de multiplicação:

- ▶ só podemos multiplicar matrizes aos pares
- ▶ devemos escolher uma **parentização** para o produto
- ▶ produto matrizes $m \times \ell$ e $\ell \times n$ faz $m \cdot \ell \cdot n$ multiplicações

Parentização

Considere um produto de matrizes

$$M = M_1 \times M_2 \times \dots \times M_i \cdots \times M_n$$

- ▶ as dimensões das matrizes são dadas por um vetor b
- ▶ a matriz M_i tem b_{i-1} linhas e b_i colunas

Ordem de multiplicação:

- ▶ só podemos multiplicar matrizes aos pares
- ▶ devemos escolher uma **parentização** para o produto
- ▶ produto matrizes $m \times \ell$ e $\ell \times n$ faz $m \cdot \ell \cdot n$ multiplicações

Exemplo de parentização

Exemplo:

- ▶ seja $M = M_1 \times M_2 \times M_3 \times M_4$ tal que $b = \langle 200, 2, 30, 20, 5 \rangle$
- ▶ as possíveis parentizações são:

$$M = (M_1 \times (M_2 \times (M_3 \times M_4))) \rightarrow 5.300 \text{ multiplicações}$$

$$M = (M_1 \times ((M_2 \times M_3) \times M_4)) \rightarrow 3.400 \text{ multiplicações}$$

$$M = ((M_1 \times M_2) \times (M_3 \times M_4)) \rightarrow 4.500 \text{ multiplicações}$$

$$M = ((M_1 \times (M_2 \times M_3)) \times M_4) \rightarrow 29.200 \text{ multiplicações}$$

$$M = (((M_1 \times M_2) \times M_3) \times M_4) \rightarrow 152.000 \text{ multiplicações}$$

- ▶ a ordem das multiplicações faz diferença!

Exemplo de parentização

Exemplo:

- ▶ seja $M = M_1 \times M_2 \times M_3 \times M_4$ tal que $b = \langle 200, 2, 30, 20, 5 \rangle$
- ▶ as possíveis parentizações são:

$$M = (M_1 \times (M_2 \times (M_3 \times M_4))) \rightarrow 5.300 \text{ multiplicações}$$

$$M = (M_1 \times ((M_2 \times M_3) \times M_4)) \rightarrow 3.400 \text{ multiplicações}$$

$$M = ((M_1 \times M_2) \times (M_3 \times M_4)) \rightarrow 4.500 \text{ multiplicações}$$

$$M = ((M_1 \times (M_2 \times M_3)) \times M_4) \rightarrow 29.200 \text{ multiplicações}$$

$$M = (((M_1 \times M_2) \times M_3) \times M_4) \rightarrow 152.000 \text{ multiplicações}$$

- ▶ a ordem das multiplicações faz diferença!

Exemplo de parentização

Exemplo:

- ▶ seja $M = M_1 \times M_2 \times M_3 \times M_4$ tal que $b = \langle 200, 2, 30, 20, 5 \rangle$
- ▶ as possíveis parentizações são:

$$M = (M_1 \times (M_2 \times (M_3 \times M_4))) \rightarrow 5.300 \text{ multiplicações}$$

$$M = (M_1 \times ((M_2 \times M_3) \times M_4)) \rightarrow 3.400 \text{ multiplicações}$$

$$M = ((M_1 \times M_2) \times (M_3 \times M_4)) \rightarrow 4.500 \text{ multiplicações}$$

$$M = (((M_1 \times (M_2 \times M_3)) \times M_4) \rightarrow 29.200 \text{ multiplicações}$$

$$M = (((M_1 \times M_2) \times M_3) \times M_4) \rightarrow 152.000 \text{ multiplicações}$$

- ▶ a ordem das multiplicações faz diferença!

Exemplo de parentização

Exemplo:

- ▶ seja $M = M_1 \times M_2 \times M_3 \times M_4$ tal que $b = \langle 200, 2, 30, 20, 5 \rangle$
- ▶ as possíveis parentizações são:

$$M = (M_1 \times (M_2 \times (M_3 \times M_4))) \rightarrow 5.300 \text{ multiplicações}$$

$$M = (M_1 \times ((M_2 \times M_3) \times M_4)) \rightarrow 3.400 \text{ multiplicações}$$

$$M = ((M_1 \times M_2) \times (M_3 \times M_4)) \rightarrow 4.500 \text{ multiplicações}$$

$$M = ((M_1 \times (M_2 \times M_3)) \times M_4) \rightarrow 29.200 \text{ multiplicações}$$

$$M = (((M_1 \times M_2) \times M_3) \times M_4) \rightarrow 152.000 \text{ multiplicações}$$

- ▶ a ordem das multiplicações faz diferença!

Multiplicação de cadeias de matrizes

Problema:

- ▶ Entrada: vetor b com dimensões de n matrizes
- ▶ Solução: parentização das matrizes
- ▶ Objetivo: **minimizar** o número de multiplicações

Testando todas as soluções viáveis

- ▶ o número de parentizações é dado por

$$P(n) = \begin{cases} 1, & n = 1 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & n > 1, \end{cases}$$

- ▶ a solução dessa recorrência é $P(n) = \Omega(4^n/n^{\frac{3}{2}})$
- ▶ um algoritmo de força bruta é **impraticável**

Multiplicação de cadeias de matrizes

Problema:

- ▶ Entrada: vetor b com dimensões de n matrizes
- ▶ Solução: parentização das matrizes
- ▶ Objetivo: **minimizar** o número de multiplicações

Testando todas as soluções viáveis

- ▶ o número de parentizações é dado por

$$P(n) = \begin{cases} 1, & n = 1 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & n > 1, \end{cases}$$

- ▶ a solução dessa recorrência é $P(n) = \Omega(4^n/n^{\frac{3}{2}})$
- ▶ um algoritmo de força bruta é **impraticável**

Multiplicação de cadeias de matrizes

Problema:

- ▶ Entrada: vetor b com dimensões de n matrizes
- ▶ Solução: parentização das matrizes
- ▶ Objetivo: **minimizar** o número de multiplicações

Testando todas as soluções viáveis

- ▶ o número de parentizações é dado por

$$P(n) = \begin{cases} 1, & n = 1 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & n > 1, \end{cases}$$

- ▶ a solução dessa recorrência é $P(n) = \Omega(4^n/n^{\frac{3}{2}})$
- ▶ um algoritmo de força bruta é **impraticável**

Multiplicação de cadeias de matrizes

Problema:

- ▶ Entrada: vetor b com dimensões de n matrizes
- ▶ Solução: parentização das matrizes
- ▶ Objetivo: **minimizar** o número de multiplicações

Testando todas as soluções viáveis

- ▶ o número de parentizações é dado por

$$P(n) = \begin{cases} 1, & n = 1 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & n > 1, \end{cases}$$

- ▶ a solução dessa recorrência é $P(n) = \Omega(4^n/n^{\frac{3}{2}})$
- ▶ um algoritmo de força bruta é **impraticável**

Multiplicação de cadeias de matrizes

Problema:

- ▶ Entrada: vetor b com dimensões de n matrizes
- ▶ Solução: parentização das matrizes
- ▶ Objetivo: **minimizar** o número de multiplicações

Testando todas as soluções viáveis

- ▶ o número de parentizações é dado por

$$P(n) = \begin{cases} 1, & n = 1 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & n > 1, \end{cases}$$

- ▶ a solução dessa recorrência é $P(n) = \Omega(4^n/n^{\frac{3}{2}})$
- ▶ um algoritmo de força bruta é **impraticável**

Multiplicação de cadeias de matrizes

Problema:

- ▶ Entrada: vetor b com dimensões de n matrizes
- ▶ Solução: parentização das matrizes
- ▶ Objetivo: **minimizar** o número de multiplicações

Testando todas as soluções viáveis

- ▶ o número de parentizações é dado por

$$P(n) = \begin{cases} 1, & n = 1 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & n > 1, \end{cases}$$

- ▶ a solução dessa recorrência é $P(n) = \Omega(4^n/n^{\frac{3}{2}})$
- ▶ um algoritmo de força bruta é **impraticável**

Multiplicação de cadeias de matrizes

Problema:

- ▶ Entrada: vetor b com dimensões de n matrizes
- ▶ Solução: parentização das matrizes
- ▶ Objetivo: **minimizar** o número de multiplicações

Testando todas as soluções viáveis

- ▶ o número de parentizações é dado por

$$P(n) = \begin{cases} 1, & n = 1 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & n > 1, \end{cases}$$

- ▶ a solução dessa recorrência é $P(n) = \Omega(4^n/n^{\frac{3}{2}})$
- ▶ um algoritmo de força bruta é **impraticável**

Multiplicação de cadeias de matrizes

Problema:

- ▶ Entrada: vetor b com dimensões de n matrizes
- ▶ Solução: parentização das matrizes
- ▶ Objetivo: **minimizar** o número de multiplicações

Testando todas as soluções viáveis

- ▶ o número de parentizações é dado por

$$P(n) = \begin{cases} 1, & n = 1 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & n > 1, \end{cases}$$

- ▶ a solução dessa recorrência é $P(n) = \Omega(4^n/n^{\frac{3}{2}})$
- ▶ um algoritmo de força bruta é **impraticável**

Encontrando uma subestrutura ótima

Considere uma **parentização ótima**

- ▶ para cada par (i, j) tal que $1 \leq i \leq j \leq n$, defina

$$M_{i,j} = M_i \times M_{i+1} \times \dots \times M_j$$

- ▶ existe k tal que a última multiplicação realizada é

$$M = M_{1,k} \times M_{k+1,n}$$

- ▶ como essa parentização é ótima, o número de multiplicações para computar $M_{i,k}$ e $M_{k+1,n}$ é mínimo

Encontramos uma **subestrutura ótima**

- ▶ os produtos $M_{i,k}$ e $M_{k+1,n}$ têm parentizações ótimas

Encontrando uma subestrutura ótima

Considere uma **parentização ótima**

- ▶ para cada par (i, j) tal que $1 \leq i \leq j \leq n$, defina

$$M_{i,j} = M_i \times M_{i+1} \times \dots \times M_j$$

- ▶ existe k tal que a última multiplicação realizada é

$$M = M_{1,k} \times M_{k+1,n}$$

- ▶ como essa parentização é ótima, o número de multiplicações para computar $M_{i,k}$ e $M_{k+1,n}$ é mínimo

Encontramos uma **subestrutura ótima**

- ▶ os produtos $M_{i,k}$ e $M_{k+1,n}$ têm parentizações ótimas

Encontrando uma subestrutura ótima

Considere uma **parentização ótima**

- ▶ para cada par (i, j) tal que $1 \leq i \leq j \leq n$, defina

$$M_{i,j} = M_i \times M_{i+1} \times \dots \times M_j$$

- ▶ existe k tal que a última multiplicação realizada é

$$M = M_{1,k} \times M_{k+1,n}$$

- ▶ como essa parentização é ótima, o número de multiplicações para computar $M_{i,k}$ e $M_{k+1,n}$ é mínimo

Encontramos uma **subestrutura ótima**

- ▶ os produtos $M_{i,k}$ e $M_{k+1,n}$ têm parentizações ótimas

Encontrando uma subestrutura ótima

Considere uma **parentização ótima**

- ▶ para cada par (i, j) tal que $1 \leq i \leq j \leq n$, defina

$$M_{i,j} = M_i \times M_{i+1} \times \dots \times M_j$$

- ▶ existe k tal que a última multiplicação realizada é

$$M = M_{1,k} \times M_{k+1,n}$$

- ▶ como essa parentização é ótima, o número de multiplicações para computar $M_{i,k}$ e $M_{k+1,n}$ é mínimo

Encontramos uma **subestrutura ótima**

- ▶ os produtos $M_{i,k}$ e $M_{k+1,n}$ têm parentizações ótimas

Encontrando uma subestrutura ótima

Considere uma **parentização ótima**

- ▶ para cada par (i, j) tal que $1 \leq i \leq j \leq n$, defina

$$M_{i,j} = M_i \times M_{i+1} \times \dots \times M_j$$

- ▶ existe k tal que a última multiplicação realizada é

$$M = M_{1,k} \times M_{k+1,n}$$

- ▶ como essa parentização é ótima, o número de multiplicações para computar $M_{i,k}$ e $M_{k+1,n}$ é mínimo

Encontramos uma **subestrutura ótima**

- ▶ os produtos $M_{i,k}$ e $M_{k+1,n}$ têm parentizações ótimas

Encontrando uma subestrutura ótima

Considere uma **parentização ótima**

- ▶ para cada par (i, j) tal que $1 \leq i \leq j \leq n$, defina

$$M_{i,j} = M_i \times M_{i+1} \times \dots \times M_j$$

- ▶ existe k tal que a última multiplicação realizada é

$$M = M_{1,k} \times M_{k+1,n}$$

- ▶ como essa parentização é ótima, o número de multiplicações para computar $M_{i,k}$ e $M_{k+1,n}$ é mínimo

Encontramos uma **subestrutura ótima**

- ▶ os produtos $M_{i,k}$ e $M_{k+1,n}$ têm parentizações ótimas

Subproblema

Definimos o seguinte **subproblema**

- ▶ considere uma tabela m com entrada para cada par (i, j)
- ▶ seja $m[i, j]$ o valor de uma parentização ótima para

$$M_i \times M_{i+1} \times \dots \times M_j$$

Podemos resolver esse subproblema recursivamente

- ▶ não são feitas multiplicações se $i = j$, então $m[i, j] = 0$
- ▶ do contrário, deve haver uma última multiplicação
- ▶ listando as possibilidades, obtemos uma **recorrência**

$$m[i, j] = \min_{i \leq k < j} \{ m[i, k] + m[k + 1, j] + b_{i-1} \cdot b_k \cdot b_j \}$$

Subproblema

Definimos o seguinte **subproblema**

- ▶ considere uma tabela m com entrada para cada par (i, j)
- ▶ seja $m[i, j]$ o valor de uma parentização ótima para

$$M_i \times M_{i+1} \times \dots \times M_j$$

Podemos resolver esse subproblema recursivamente

- ▶ não são feitas multiplicações se $i = j$, então $m[i, j] = 0$
- ▶ do contrário, deve haver uma última multiplicação
- ▶ listando as possibilidades, obtemos uma **recorrência**

$$m[i, j] = \min_{i \leq k < j} \{ m[i, k] + m[k + 1, j] + b_{i-1} \cdot b_k \cdot b_j \}$$

Subproblema

Definimos o seguinte **subproblema**

- ▶ considere uma tabela m com entrada para cada par (i, j)
- ▶ seja $m[i, j]$ o valor de uma parentização ótima para

$$M_i \times M_{i+1} \times \dots \times M_j$$

Podemos resolver esse subproblema recursivamente

- ▶ não são feitas multiplicações se $i = j$, então $m[i, j] = 0$
- ▶ do contrário, deve haver uma última multiplicação
- ▶ listando as possibilidades, obtemos uma **recorrência**

$$m[i, j] = \min_{i \leq k < j} \{ m[i, k] + m[k + 1, j] + b_{i-1} \cdot b_k \cdot b_j \}$$

Subproblema

Definimos o seguinte **subproblema**

- ▶ considere uma tabela m com entrada para cada par (i, j)
- ▶ seja $m[i, j]$ o valor de uma parentização ótima para

$$M_i \times M_{i+1} \times \dots \times M_j$$

Podemos resolver esse subproblema recursivamente

- ▶ não são feitas multiplicações se $i = j$, então $m[i, j] = 0$
- ▶ do contrário, deve haver uma última multiplicação
- ▶ listando as possibilidades, obtemos uma **recorrência**

$$m[i, j] = \min_{i \leq k < j} \{ m[i, k] + m[k + 1, j] + b_{i-1} \cdot b_k \cdot b_j \}$$

Subproblema

Definimos o seguinte **subproblema**

- ▶ considere uma tabela m com entrada para cada par (i, j)
- ▶ seja $m[i, j]$ o valor de uma parentização ótima para

$$M_i \times M_{i+1} \times \dots \times M_j$$

Podemos resolver esse subproblema recursivamente

- ▶ não são feitas multiplicações se $i = j$, então $m[i, j] = 0$
- ▶ do contrário, deve haver uma última multiplicação
- ▶ listando as possibilidades, obtemos uma **recorrência**

$$m[i, j] = \min_{i \leq k < j} \{ m[i, k] + m[k + 1, j] + b_{i-1} \cdot b_k \cdot b_j \}$$

Subproblema

Definimos o seguinte **subproblema**

- ▶ considere uma tabela m com entrada para cada par (i, j)
- ▶ seja $m[i, j]$ o valor de uma parentização ótima para

$$M_i \times M_{i+1} \times \dots \times M_j$$

Podemos resolver esse subproblema recursivamente

- ▶ não são feitas multiplicações se $i = j$, então $m[i, j] = 0$
- ▶ do contrário, deve haver uma última multiplicação
- ▶ listando as possibilidades, obtemos uma **recorrência**

$$m[i, j] = \min_{i \leq k < j} \{ m[i, k] + m[k + 1, j] + b_{i-1} \cdot b_k \cdot b_j \}$$

Subproblema

Definimos o seguinte **subproblema**

- ▶ considere uma tabela m com entrada para cada par (i, j)
- ▶ seja $m[i, j]$ o valor de uma parentização ótima para

$$M_i \times M_{i+1} \times \dots \times M_j$$

Podemos resolver esse subproblema recursivamente

- ▶ não são feitas multiplicações se $i = j$, então $m[i, j] = 0$
- ▶ do contrário, deve haver uma última multiplicação
- ▶ listando as possibilidades, obtemos uma **recorrência**

$$m[i, j] = \min_{i \leq k < j} \{ m[i, k] + m[k + 1, j] + b_{i-1} \cdot b_k \cdot b_j \}$$

Algoritmo recursivo

MINIMO-MULTIPLICACOES-RECURSIVO(b, i, j)

```
1  se  $i = j$  então
2     $m[i, j] \leftarrow 0$ 
3  senão
4     $m[i, j] \leftarrow \infty$ 
5    para  $k \leftarrow i$  até  $j - 1$  faça
6       $q \leftarrow$  MINIMO-MULTIPLICACOES-RECURSIVO( $b, i, k$ )
          + MINIMO-MULTIPLICACOES-RECURSIVO( $b, k + 1, j$ )
          +  $b[i - 1] \cdot b[k] \cdot b[j]$ 
7      se  $m > q$  então
8         $m[i, j] \leftarrow q$ 
9         $s[i, j] \leftarrow k$ 
10  devolva  $m[i, j]$ 
```

- ▶ $s[i, j]$ guarda o índice para a última multiplicação de $M_{i,j}$
- ▶ a chamada inicial é MINIMO-MULTIPLICACOES-RECURSIVO($b, 1, n$)

Algoritmo recursivo

MINIMO-MULTIPLICACOES-RECURSIVO(b, i, j)

```
1  se  $i = j$  então
2     $m[i, j] \leftarrow 0$ 
3  senão
4     $m[i, j] \leftarrow \infty$ 
5    para  $k \leftarrow i$  até  $j - 1$  faça
6       $q \leftarrow$  MINIMO-MULTIPLICACOES-RECURSIVO( $b, i, k$ )
          + MINIMO-MULTIPLICACOES-RECURSIVO( $b, k + 1, j$ )
          +  $b[i - 1] \cdot b[k] \cdot b[j]$ 
7      se  $m > q$  então
8         $m[i, j] \leftarrow q$ 
9         $s[i, j] \leftarrow k$ 
10  devolva  $m[i, j]$ 
```

- ▶ $s[i, j]$ guarda o índice para a última multiplicação de $M_{i, j}$
- ▶ a chamada inicial é MINIMO-MULTIPLICACOES-RECURSIVO($b, 1, n$)

Algoritmo recursivo

MINIMO-MULTIPLICACOES-RECURSIVO(b, i, j)

```
1  se  $i = j$  então
2     $m[i, j] \leftarrow 0$ 
3  senão
4     $m[i, j] \leftarrow \infty$ 
5    para  $k \leftarrow i$  até  $j - 1$  faça
6       $q \leftarrow$  MINIMO-MULTIPLICACOES-RECURSIVO( $b, i, k$ )
          + MINIMO-MULTIPLICACOES-RECURSIVO( $b, k + 1, j$ )
          +  $b[i - 1] \cdot b[k] \cdot b[j]$ 
7      se  $m > q$  então
8         $m[i, j] \leftarrow q$ 
9         $s[i, j] \leftarrow k$ 
10  devolva  $m[i, j]$ 
```

- ▶ $s[i, j]$ guarda o índice para a última multiplicação de $M_{i, j}$
- ▶ a chamada inicial é MINIMO-MULTIPLICACOES-RECURSIVO($b, 1, n$)

Recuperando a solução ótima

- ▶ a função anterior devolve apenas o valor da solução
- ▶ mas a tabela s induz uma **parentização ótima**

MULTIPLICA-MATRIZES(M, s, i, j)

```
1  se  $i = j$  então
2      devolva  $M_i$ 
3  senão
4       $X \leftarrow$  MULTIPLICA-MATRIZES( $M, s, i, s[i, j]$ )
5       $Y \leftarrow$  MULTIPLICA-MATRIZES( $M, s, s[i, j] + 1, j$ )
6      devolva Multiplica( $X, Y, b[i - 1], b[s[i, j]], b[j]$ )
```

Recuperando a solução ótima

- ▶ a função anterior devolve apenas o valor da solução
- ▶ mas a tabela s induz uma **parentização ótima**

MULTIPLICA-MATRIZES(M, s, i, j)

1 se $i = j$ então

2 devolva M_i

3 senão

4 $X \leftarrow$ MULTIPLICA-MATRIZES($M, s, i, s[i, j]$)

5 $Y \leftarrow$ MULTIPLICA-MATRIZES($M, s, s[i, j] + 1, j$)

6 devolva *Multiplica*($X, Y, b[i - 1], b[s[i, j]], b[j]$)

Recuperando a solução ótima

- ▶ a função anterior devolve apenas o valor da solução
- ▶ mas a tabela s induz uma **parentização ótima**

MULTIPLICA-MATRIZES(M, s, i, j)

- 1 se $i = j$ então
- 2 devolva M_i
- 3 senão
- 4 $X \leftarrow \text{MULTIPLICA-MATRIZES}(M, s, i, s[i, j])$
- 5 $Y \leftarrow \text{MULTIPLICA-MATRIZES}(M, s, s[i, j] + 1, j)$
- 6 devolva $\text{Multiplica}(X, Y, b[i - 1], b[s[i, j]], b[j])$

Complexidade do algoritmo recursivo

Seja $n = j - i + 1$ o número de matrizes

- ▶ o tempo de execução é dado pela recorrência

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ \sum_{k=1}^{n-1} [T(k) + T(n-k)] + \Theta(n) & \text{se } n > 1 \end{cases}$$

- ▶ podemos mostrar que $T(n) = \Omega(2^n)$
- ▶ isso ainda é impraticável

Complexidade do algoritmo recursivo

Seja $n = j - i + 1$ o número de matrizes

- ▶ o tempo de execução é dado pela recorrência

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ \sum_{k=1}^{n-1} [T(k) + T(n-k)] + \Theta(n) & \text{se } n > 1 \end{cases}$$

- ▶ podemos mostrar que $T(n) = \Omega(2^n)$
- ▶ isso ainda é impraticável

Complexidade do algoritmo recursivo

Seja $n = j - i + 1$ o número de matrizes

- ▶ o tempo de execução é dado pela recorrência

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ \sum_{k=1}^{n-1} [T(k) + T(n-k)] + \Theta(n) & \text{se } n > 1 \end{cases}$$

- ▶ podemos mostrar que $T(n) = \Omega(2^n)$
- ▶ isso ainda é impraticável

Complexidade do algoritmo recursivo

Seja $n = j - i + 1$ o número de matrizes

- ▶ o tempo de execução é dado pela recorrência

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ \sum_{k=1}^{n-1} [T(k) + T(n-k)] + \Theta(n) & \text{se } n > 1 \end{cases}$$

- ▶ podemos mostrar que $T(n) = \Omega(2^n)$
- ▶ isso ainda é **impraticável**

Complexidade do algoritmo recursivo (cont)

Analizando com calma

- ▶ o algoritmo **recalcula** o mesmo valor $m[i,j]$ várias vezes
- ▶ por exemplo, se $n = 4$, os valores $m[1,2]$, $m[2,3]$ e $m[3,4]$ são computados duas vezes

Melhorando

- ▶ o número de pares (i,j) distintos é limitado por $O(n^2)$
- ▶ podemos guardar todos os valores $m[i,j]$ em uma tabela
- ▶ e resolver cada subproblema apenas uma vez

Complexidade do algoritmo recursivo (cont)

Analisando com calma

- ▶ o algoritmo **recalcula** o mesmo valor $m[i,j]$ várias vezes
- ▶ por exemplo, se $n = 4$, os valores $m[1,2]$, $m[2,3]$ e $m[3,4]$ são computados duas vezes

Melhorando

- ▶ o número de pares (i,j) distintos é limitado por $O(n^2)$
- ▶ podemos guardar todos os valores $m[i,j]$ em uma tabela
- ▶ e resolver cada subproblema apenas uma vez

Complexidade do algoritmo recursivo (cont)

Analisando com calma

- ▶ o algoritmo **recalcula** o mesmo valor $m[i,j]$ várias vezes
- ▶ por exemplo, se $n = 4$, os valores $m[1,2]$, $m[2,3]$ e $m[3,4]$ são computados duas vezes

Melhorando

- ▶ o número de pares (i,j) distintos é limitado por $O(n^2)$
- ▶ podemos guardar todos os valores $m[i,j]$ em uma tabela
- ▶ e resolver cada subproblema apenas uma vez

Complexidade do algoritmo recursivo (cont)

Analisando com calma

- ▶ o algoritmo **recalcula** o mesmo valor $m[i,j]$ várias vezes
- ▶ por exemplo, se $n = 4$, os valores $m[1,2]$, $m[2,3]$ e $m[3,4]$ são computados duas vezes

Melhorando

- ▶ o número de pares (i,j) distintos é limitado por $O(n^2)$
- ▶ podemos guardar todos os valores $m[i,j]$ em uma tabela
- ▶ e resolver cada subproblema apenas uma vez

Complexidade do algoritmo recursivo (cont)

Analisando com calma

- ▶ o algoritmo **recalcula** o mesmo valor $m[i,j]$ várias vezes
- ▶ por exemplo, se $n = 4$, os valores $m[1,2]$, $m[2,3]$ e $m[3,4]$ são computados duas vezes

Melhorando

- ▶ o número de pares (i,j) distintos é limitado por $O(n^2)$
- ▶ podemos guardar todos os valores $m[i,j]$ em uma tabela
- ▶ e resolver cada subproblema apenas uma vez

Complexidade do algoritmo recursivo (cont)

Analisando com calma

- ▶ o algoritmo **recalcula** o mesmo valor $m[i,j]$ várias vezes
- ▶ por exemplo, se $n = 4$, os valores $m[1,2]$, $m[2,3]$ e $m[3,4]$ são computados duas vezes

Melhorando

- ▶ o número de pares (i,j) distintos é limitado por $O(n^2)$
- ▶ podemos guardar todos os valores $m[i,j]$ em uma tabela
- ▶ e resolver cada subproblema apenas uma vez

Complexidade do algoritmo recursivo (cont)

Analisando com calma

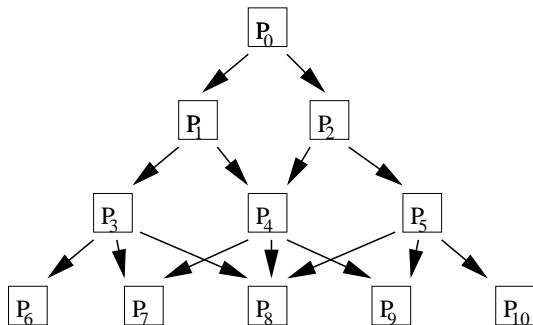
- ▶ o algoritmo **recalcula** o mesmo valor $m[i,j]$ várias vezes
- ▶ por exemplo, se $n = 4$, os valores $m[1,2]$, $m[2,3]$ e $m[3,4]$ são computados duas vezes

Melhorando

- ▶ o número de pares (i,j) distintos é limitado por $O(n^2)$
- ▶ podemos guardar todos os valores $m[i,j]$ em uma tabela
- ▶ e resolver cada subproblema apenas uma vez

Recalculando os mesmos subproblemas

Considere a execução de um algoritmo recursivo

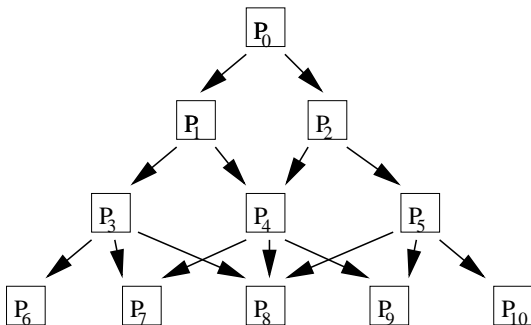


Nessa árvore de chamadas

- ▶ P_4 é resolvido 2 vezes
- ▶ P_7 é resolvido 3 vezes
- ▶ P_8 é resolvido 4 vezes
- ▶ P_9 é resolvido 3 vezes

Recalculando os mesmos subproblemas

Considere a execução de um algoritmo recursivo

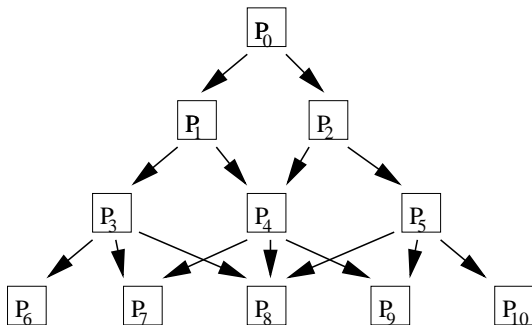


Nessa árvore de chamadas

- ▶ P_4 é resolvido 2 vezes
- ▶ P_7 é resolvido 3 vezes
- ▶ P_8 é resolvido 4 vezes
- ▶ P_9 é resolvido 3 vezes

Recalculando os mesmos subproblemas

Considere a execução de um algoritmo recursivo

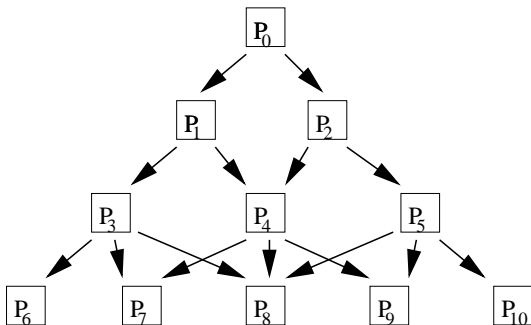


Nessa árvore de chamadas

- ▶ P_4 é resolvido 2 vezes
- ▶ P_7 é resolvido 3 vezes
- ▶ P_8 é resolvido 4 vezes
- ▶ P_9 é resolvido 3 vezes

Recalculando os mesmos subproblemas

Considere a execução de um algoritmo recursivo

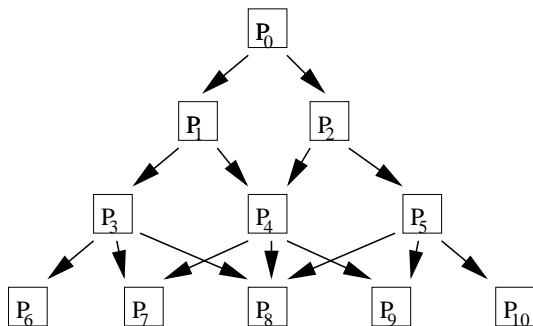


Nessa árvore de chamadas

- ▶ P_4 é resolvido 2 vezes
- ▶ P_7 é resolvido 3 vezes
- ▶ P_8 é resolvido 4 vezes
- ▶ P_9 é resolvido 3 vezes

Recalculando os mesmos subproblemas

Considere a execução de um algoritmo recursivo

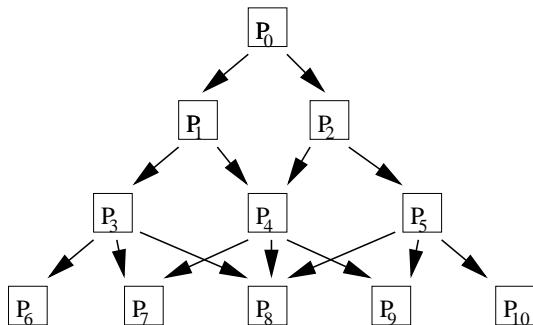


Nessa árvore de chamadas

- ▶ P_4 é resolvido 2 vezes
- ▶ P_7 é resolvido 3 vezes
- ▶ P_8 é resolvido 4 vezes
- ▶ P_9 é resolvido 3 vezes

Recalculando os mesmos subproblemas

Considere a execução de um algoritmo recursivo



Nessa árvore de chamadas

- ▶ P_4 é resolvido 2 vezes
- ▶ P_7 é resolvido 3 vezes
- ▶ P_8 é resolvido 4 vezes
- ▶ P_9 é resolvido 3 vezes

Evitamos o recálculo de subproblemas de duas maneiras

1. Memorização

- ▶ mantemos a estrutura recursiva do algoritmo
- ▶ guardamos os valores computados em tabela
- ▶ devolvemos valores já conhecidos antes da chamada recursiva

2. Programação dinâmica:

- ▶ criamos uma tabela com entrada para cada subproblema
- ▶ inicializamos as entradas correspondentes a casos básicos
- ▶ preenchemos o restante da tabela com uma recorrência
- ▶ a ordem de preenchimento deve obedecer a dependência entre subproblemas

Evitamos o recálculo de subproblemas de duas maneiras

1. Memorização

- ▶ mantemos a estrutura recursiva do algoritmo
- ▶ guardamos os valores computados em tabela
- ▶ devolvemos valores já conhecidos antes da chamada recursiva

2. Programação dinâmica:

- ▶ criamos uma tabela com entrada para cada subproblema
- ▶ inicializamos as entradas correspondentes a casos básicos
- ▶ preenchemos o restante da tabela com uma recorrência
- ▶ a ordem de preenchimento deve obedecer a dependência entre subproblemas

Evitamos o recálculo de subproblemas de duas maneiras

1. Memorização

- ▶ mantemos a estrutura recursiva do algoritmo
- ▶ guardamos os valores computados em tabela
- ▶ devolvemos valores já conhecidos antes da chamada recursiva

2. Programação dinâmica:

- ▶ criamos uma tabela com entrada para cada subproblema
- ▶ inicializamos as entradas correspondentes a casos básicos
- ▶ preenchemos o restante da tabela com uma recorrência
- ▶ a ordem de preenchimento deve obedecer a dependência entre subproblemas

Evitamos o recálculo de subproblemas de duas maneiras

1. Memorização

- ▶ mantemos a estrutura recursiva do algoritmo
- ▶ guardamos os valores computados em tabela
- ▶ devolvemos valores já conhecidos antes da chamada recursiva

2. Programação dinâmica:

- ▶ criamos uma tabela com entrada para cada subproblema
- ▶ inicializamos as entradas correspondentes a casos básicos
- ▶ preenchemos o restante da tabela com uma recorrência
- ▶ a ordem de preenchimento deve obedecer a dependência entre subproblemas

Evitamos o recálculo de subproblemas de duas maneiras

1. Memorização

- ▶ mantemos a estrutura recursiva do algoritmo
- ▶ guardamos os valores computados em tabela
- ▶ devolvemos valores já conhecidos antes da chamada recursiva

2. Programação dinâmica:

- ▶ criamos uma tabela com entrada para cada subproblema
- ▶ inicializamos as entradas correspondentes a casos básicos
- ▶ preenchemos o restante da tabela com uma recorrência
- ▶ a ordem de preenchimento deve obedecer a dependência entre subproblemas

Memorização x programação dinâmica

Evitamos o recálculo de subproblemas de duas maneiras

1. Memorização

- ▶ mantemos a estrutura recursiva do algoritmo
- ▶ guardamos os valores computados em tabela
- ▶ devolvemos valores já conhecidos antes da chamada recursiva

2. Programação dinâmica:

- ▶ criamos uma tabela com entrada para cada subproblema
- ▶ inicializamos as entradas correspondentes a casos básicos
- ▶ preenchemos o restante da tabela com uma **recorrência**
- ▶ a ordem de preenchimento deve obedecer a dependência entre subproblemas

Memorização x programação dinâmica

Evitamos o recálculo de subproblemas de duas maneiras

1. Memorização

- ▶ mantemos a estrutura recursiva do algoritmo
- ▶ guardamos os valores computados em tabela
- ▶ devolvemos valores já conhecidos antes da chamada recursiva

2. Programação dinâmica:

- ▶ criamos uma tabela com entrada para cada subproblema
- ▶ inicializamos as entradas correspondentes a casos básicos
- ▶ preenchemos o restante da tabela com uma **recorrência**
- ▶ a ordem de preenchimento deve obedecer a dependência entre subproblemas

Memorização x programação dinâmica

Evitamos o recálculo de subproblemas de duas maneiras

1. Memorização

- ▶ mantemos a estrutura recursiva do algoritmo
- ▶ guardamos os valores computados em tabela
- ▶ devolvemos valores já conhecidos antes da chamada recursiva

2. Programação dinâmica:

- ▶ criamos uma tabela com entrada para cada subproblema
- ▶ inicializamos as entradas correspondentes a casos básicos
- ▶ preenchemos o restante da tabela com uma **recorrência**
- ▶ a ordem de preenchimento deve obedecer a dependência entre subproblemas

Memorização x programação dinâmica

Evitamos o recálculo de subproblemas de duas maneiras

1. Memorização

- ▶ mantemos a estrutura recursiva do algoritmo
- ▶ guardamos os valores computados em tabela
- ▶ devolvemos valores já conhecidos antes da chamada recursiva

2. Programação dinâmica:

- ▶ criamos uma tabela com entrada para cada subproblema
- ▶ inicializamos as entradas correspondentes a casos básicos
- ▶ preenchemos o restante da tabela com uma **recorrência**
- ▶ a ordem de preenchimento deve obedecer a dependência entre subproblemas

Memorização x programação dinâmica

Evitamos o recálculo de subproblemas de duas maneiras

1. Memorização

- ▶ mantemos a estrutura recursiva do algoritmo
- ▶ guardamos os valores computados em tabela
- ▶ devolvemos valores já conhecidos antes da chamada recursiva

2. Programação dinâmica:

- ▶ criamos uma tabela com entrada para cada subproblema
- ▶ inicializamos as entradas correspondentes a casos básicos
- ▶ preenchemos o restante da tabela com uma **recorrência**
- ▶ a ordem de preenchimento deve obedecer a dependência entre subproblemas

Algoritmo de memorização

MINIMO-MULTIPLICACOES-MEMORIZADO(b, i, j)

```
1  se  $m[i, j]$  não está definido então
2    se  $i = j$  então
3       $m[i, j] \leftarrow 0$ 
4    senão
5       $m[i, j] \leftarrow \infty$ 
6      para  $k \leftarrow i$  até  $j - 1$  faça
7         $q \leftarrow$  MINIMO-MULTIPLICACOES-MEMORIZADO( $b, i, k$ )
          + MINIMO-MULTIPLICACOES-MEMORIZADO( $b, k + 1, j$ )
          +  $b[i - 1] \cdot b[k] \cdot b[j]$ 
8        se  $m > q$  então
9           $m[i, j] \leftarrow q$ 
10          $s[i, j] \leftarrow k$ 
11  devolva  $m[i, j]$ 
```

Algoritmo de programação dinâmica

- ▶ denotamos por u o tamanho da cadeia do subproblema
- ▶ preenchemos a tabela em ordem crescente de tamanho

MINIMO-MULTIPLICACOES(b)

```
1  para  $i \leftarrow 1$  até  $n$  faça
2       $m[i, i] \leftarrow 0$ 
3  para  $u \leftarrow 1$  até  $n - 1$  faça
4      para  $i \leftarrow 1$  até  $n - u$  faça
5           $j \leftarrow i + u$ 
6           $m[i, j] \leftarrow \infty$ 
7          para  $k \leftarrow i$  até  $j - 1$  faça
8               $q \leftarrow m[i, k] + m[k + 1, j] + b[i - 1] \cdot b[k] \cdot b[j]$ 
9              se  $q < m[i, j]$  então
10                  $m[i, j] \leftarrow q$ 
11                  $s[i, j] \leftarrow k$ 
12  devolva  $m[1, n]$ 
```

Algoritmo de programação dinâmica

- ▶ denotamos por u o tamanho da cadeia do subproblema
- ▶ preenchemos a tabela em ordem crescente de tamanho

MINIMO-MULTIPLICACOES(b)

```
1  para  $i \leftarrow 1$  até  $n$  faça
2       $m[i, i] \leftarrow 0$ 
3  para  $u \leftarrow 1$  até  $n - 1$  faça
4      para  $i \leftarrow 1$  até  $n - u$  faça
5           $j \leftarrow i + u$ 
6           $m[i, j] \leftarrow \infty$ 
7          para  $k \leftarrow i$  até  $j - 1$  faça
8               $q \leftarrow m[i, k] + m[k + 1, j] + b[i - 1] \cdot b[k] \cdot b[j]$ 
9              se  $q < m[i, j]$  então
10                  $m[i, j] \leftarrow q$ 
11                  $s[i, j] \leftarrow k$ 
12  devolva  $m[1, n]$ 
```

Algoritmo de programação dinâmica

- ▶ denotamos por u o tamanho da cadeia do subproblema
- ▶ preenchemos a tabela em ordem crescente de tamanho

MINIMO-MULTIPLICACOES(b)

```
1  para  $i \leftarrow 1$  até  $n$  faça
2       $m[i, i] \leftarrow 0$ 
3  para  $u \leftarrow 1$  até  $n - 1$  faça
4      para  $i \leftarrow 1$  até  $n - u$  faça
5           $j \leftarrow i + u$ 
6           $m[i, j] \leftarrow \infty$ 
7          para  $k \leftarrow i$  até  $j - 1$  faça
8               $q \leftarrow m[i, k] + m[k + 1, j] + b[i - 1] \cdot b[k] \cdot b[j]$ 
9              se  $q < m[i, j]$  então
10                  $m[i, j] \leftarrow q$ 
11                  $s[i, j] \leftarrow k$ 
12  devolva  $m[1, n]$ 
```

Exemplo de tabela

			i	$i+1$	$i+2$	$i+3$	j
i							
$i+1$							
$i+2$							
$i+3$							
j							

Exemplo de preenchimento

	1	2	3	4
1	0			
2		0		
3			0	
4				0

m

	1	2	3	4
1	-			
2		-		
3			-	
4				-

s

Exemplo de preenchimento

	1	2	3	4
1	0	12000		
2		0	1200	
3			0	3000
4				0

m

	1	2	3	4
1	-	1		
2		-	2	
3			-	3
4				-

s

{ 200, 2, 30, 20, 5 }

Exemplo de preenchimento

	1	2	3	4
1	0	12000	9200	
2		0	1200	
3			0	3000
4				0

m

	1	2	3	4
1	-	1	1	
2		-	2	
3			-	3
4				-

s

{ 200, 2, 30, 20, 5 }

$$b_0 * b_1 * b_3 = 200 * 2 * 20 = 8000$$

$$b_0 * b_2 * b_3 = 200 * 30 * 20 = 120000$$

Exemplo de preenchimento

	1	2	3	4
1	0	12000	9200	
2		0	1200	1400
3			0	3000
4				0

m

	1	2	3	4
1	-	1	1	
2		-	2	3
3			-	3
4				-

s

{ 200, 2, 30, 20, 5 }

$$b1 * b2 * b4 = 2 * 30 * 5 = 300$$

$$b1 * b3 * b4 = 2 * 20 * 5 = 200$$

Exemplo de preenchimento

	1	2	3	4
1	0	12000	9200	3400
2		0	1200	1400
3			0	3400
4				0

m

$$b_0 * b_1 * b_4 = 200 * 2 * 5 = 2000$$

$$b_0 * b_2 * b_4 = 200 * 30 * 5 = 30000$$

$$b_0 * b_3 * b_4 = 200 * 20 * 5 = 20000$$

	1	2	3	4
1	-	1	1	1
2		-	2	3
3			-	3
4				-

s

{ 200, 2, 30, 20, 5 }

Exemplo de preenchimento

	1	2	3	4
1	0	12000	9200	3400
2		0	1200	1400
3			0	3000
4				0

m

	1	2	3	4
1	-	1	1	1
2		-	2	3
3			-	3
4				-

s

M1 ((M2 . M3) . M4)

Complexidade da programação dinâmica

Análise

- ▶ o algoritmo preenche cada entrada (i,j) uma vez
- ▶ o número de pares diferentes na tabela é $O(n^2)$
- ▶ preencher cada entrada leva tempo $O(n)$
- ▶ o tempo total é limitado número de pares \times tempo por par
- ▶ além disso, usamos uma matriz $n \times n$ para a tabela

Complexidade

- ▶ o tempo gasto pelo algoritmo é $O(n^3)$
- ▶ a memória usada pelo algoritmo é $O(n^2)$

Complexidade da programação dinâmica

Análise

- ▶ o algoritmo preenche cada entrada (i,j) uma vez
- ▶ o número de pares diferentes na tabela é $O(n^2)$
- ▶ preencher cada entrada leva tempo $O(n)$
- ▶ o tempo total é limitado número de pares \times tempo por par
- ▶ além disso, usamos uma matriz $n \times n$ para a tabela

Complexidade

- ▶ o tempo gasto pelo algoritmo é $O(n^3)$
- ▶ a memória usada pelo algoritmo é $O(n^2)$

Complexidade da programação dinâmica

Análise

- ▶ o algoritmo preenche cada entrada (i,j) uma vez
- ▶ o número de pares diferentes na tabela é $O(n^2)$
- ▶ preencher cada entrada leva tempo $O(n)$
- ▶ o tempo total é limitado número de pares \times tempo por par
- ▶ além disso, usamos uma matriz $n \times n$ para a tabela

Complexidade

- ▶ o tempo gasto pelo algoritmo é $O(n^3)$
- ▶ a memória usada pelo algoritmo é $O(n^2)$

Complexidade da programação dinâmica

Análise

- ▶ o algoritmo preenche cada entrada (i,j) uma vez
- ▶ o número de pares diferentes na tabela é $O(n^2)$
- ▶ preencher cada entrada leva tempo $O(n)$
- ▶ o tempo total é limitado número de pares \times tempo por par
- ▶ além disso, usamos uma matriz $n \times n$ para a tabela

Complexidade

- ▶ o tempo gasto pelo algoritmo é $O(n^3)$
- ▶ a memória usada pelo algoritmo é $O(n^2)$

Complexidade da programação dinâmica

Análise

- ▶ o algoritmo preenche cada entrada (i,j) uma vez
- ▶ o número de pares diferentes na tabela é $O(n^2)$
- ▶ preencher cada entrada leva tempo $O(n)$
- ▶ o tempo total é limitado número de pares \times tempo por par
- ▶ além disso, usamos uma matriz $n \times n$ para a tabela

Complexidade

- ▶ o tempo gasto pelo algoritmo é $O(n^3)$
- ▶ a memória usada pelo algoritmo é $O(n^2)$

Complexidade da programação dinâmica

Análise

- ▶ o algoritmo preenche cada entrada (i,j) uma vez
- ▶ o número de pares diferentes na tabela é $O(n^2)$
- ▶ preencher cada entrada leva tempo $O(n)$
- ▶ o tempo total é limitado número de pares \times tempo por par
- ▶ além disso, usamos uma matriz $n \times n$ para a tabela

Complexidade

- ▶ o tempo gasto pelo algoritmo é $O(n^3)$
- ▶ a memória usada pelo algoritmo é $O(n^2)$

Complexidade da programação dinâmica

Análise

- ▶ o algoritmo preenche cada entrada (i,j) uma vez
- ▶ o número de pares diferentes na tabela é $O(n^2)$
- ▶ preencher cada entrada leva tempo $O(n)$
- ▶ o tempo total é limitado número de pares \times tempo por par
- ▶ além disso, usamos uma matriz $n \times n$ para a tabela

Complexidade

- ▶ o tempo gasto pelo algoritmo é $O(n^3)$
- ▶ a memória usada pelo algoritmo é $O(n^2)$

Complexidade da programação dinâmica

Análise

- ▶ o algoritmo preenche cada entrada (i,j) uma vez
- ▶ o número de pares diferentes na tabela é $O(n^2)$
- ▶ preencher cada entrada leva tempo $O(n)$
- ▶ o tempo total é limitado número de pares \times tempo por par
- ▶ além disso, usamos uma matriz $n \times n$ para a tabela

Complexidade

- ▶ o tempo gasto pelo algoritmo é $O(n^3)$
- ▶ a memória usada pelo algoritmo é $O(n^2)$

Complexidade da programação dinâmica

Análise

- ▶ o algoritmo preenche cada entrada (i,j) uma vez
- ▶ o número de pares diferentes na tabela é $O(n^2)$
- ▶ preencher cada entrada leva tempo $O(n)$
- ▶ o tempo total é limitado número de pares \times tempo por par
- ▶ além disso, usamos uma matriz $n \times n$ para a tabela

Complexidade

- ▶ o tempo gasto pelo algoritmo é $O(n^3)$
- ▶ a memória usada pelo algoritmo é $O(n^2)$

Programação dinâmica

- ▶ Problema da mochila binário

Problema do ladrão

Um ladrão vai roubar uma casa

- ▶ irá colocar alguns itens na mochila
- ▶ mas a mochila tem um limite de peso W

Ele tem n itens disponíveis

- ▶ cada item i tem um peso w_i
- ▶ cada item i tem um valor c_i

Pergunta: quais itens escolher para maximizar o valor?

Problema do ladrão

Um ladrão vai roubar uma casa

- ▶ irá colocar alguns itens na mochila
- ▶ mas a mochila tem um limite de peso W

Ele tem n itens disponíveis

- ▶ cada item i tem um peso w_i
- ▶ cada item i tem um valor c_i

Pergunta: quais itens escolher para maximizar o valor?

Problema do ladrão

Um ladrão vai roubar uma casa

- ▶ irá colocar alguns itens na mochila
- ▶ mas a mochila tem um limite de peso W

Ele tem n itens disponíveis

- ▶ cada item i tem um peso w_i
- ▶ cada item i tem um valor c_i

Pergunta: quais itens escolher para maximizar o valor?

Problema do ladrão

Um ladrão vai roubar uma casa

- ▶ irá colocar alguns itens na mochila
- ▶ mas a mochila tem um limite de peso W

Ele tem n itens disponíveis

- ▶ cada item i tem um peso w_i
- ▶ cada item i tem um valor c_i

Pergunta: quais itens escolher para maximizar o valor?

Problema do ladrão

Um ladrão vai roubar uma casa

- ▶ irá colocar alguns itens na mochila
- ▶ mas a mochila tem um limite de peso W

Ele tem n itens disponíveis

- ▶ cada item i tem um peso w_i
- ▶ cada item i tem um valor c_i

Pergunta: quais itens escolher para maximizar o valor?

Problema do ladrão

Um ladrão vai roubar uma casa

- ▶ irá colocar alguns itens na mochila
- ▶ mas a mochila tem um limite de peso W

Ele tem n itens disponíveis

- ▶ cada item i tem um peso w_i
- ▶ cada item i tem um valor c_i

Pergunta: quais itens escolher para maximizar o valor?

Problema do ladrão

Um ladrão vai roubar uma casa

- ▶ irá colocar alguns itens na mochila
- ▶ mas a mochila tem um limite de peso W

Ele tem n itens disponíveis

- ▶ cada item i tem um peso w_i
- ▶ cada item i tem um valor c_i

Pergunta: quais itens escolher para maximizar o valor?

Problema da mochila binário

Problema:

- ▶ Entrada:
 - ▶ inteiro não negativo W representando a capacidade
 - ▶ vetor de inteiros não negativos w representando pesos
 - ▶ vetor de inteiros não negativos c representando valores
- ▶ Solução:
 - ▶ subconjunto $I \subseteq \{1, 2, \dots, n\}$ tal que $\sum_{i \in I} w_i \leq W$
- ▶ Objetivo:
 - ▶ maximizar $\sum_{i \in I} c_i$

Podemos fazer algumas suposições

1. $\sum_{i=1}^n w_i > W$
2. $0 < w_i \leq W$ para todo $i = 1, \dots, n$

Problema da mochila binário

Problema:

▶ Entrada:

- ▶ inteiro não negativo W representando a capacidade
- ▶ vetor de inteiros não negativos w representando pesos
- ▶ vetor de inteiros não negativos c representando valores

▶ Solução:

- ▶ subconjunto $I \subseteq \{1, 2, \dots, n\}$ tal que $\sum_{i \in I} w_i \leq W$

▶ Objetivo:

- ▶ maximizar $\sum_{i \in I} c_i$

Podemos fazer algumas suposições

1. $\sum_{i=1}^n w_i > W$
2. $0 < w_i \leq W$ para todo $i = 1, \dots, n$

Problema da mochila binário

Problema:

- ▶ Entrada:
 - ▶ inteiro não negativo W representando a capacidade
 - ▶ vetor de inteiros não negativos w representando pesos
 - ▶ vetor de inteiros não negativos c representando valores
- ▶ Solução:
 - ▶ subconjunto $I \subseteq \{1, 2, \dots, n\}$ tal que $\sum_{i \in I} w_i \leq W$
- ▶ Objetivo:
 - ▶ maximizar $\sum_{i \in I} c_i$

Podemos fazer algumas suposições

1. $\sum_{i=1}^n w_i > W$
2. $0 < w_i \leq W$ para todo $i = 1, \dots, n$

Problema da mochila binário

Problema:

- ▶ Entrada:
 - ▶ inteiro não negativo W representando a capacidade
 - ▶ vetor de inteiros não negativos w representando pesos
 - ▶ vetor de inteiros não negativos c representando valores
- ▶ Solução:
 - ▶ subconjunto $I \subseteq \{1, 2, \dots, n\}$ tal que $\sum_{i \in I} w_i \leq W$
- ▶ Objetivo:
 - ▶ maximizar $\sum_{i \in I} c_i$

Podemos fazer algumas suposições

1. $\sum_{i=1}^n w_i > W$
2. $0 < w_i \leq W$ para todo $i = 1, \dots, n$

Problema da mochila binário

Problema:

- ▶ Entrada:
 - ▶ inteiro não negativo W representando a capacidade
 - ▶ vetor de inteiros não negativos w representando pesos
 - ▶ vetor de inteiros não negativos c representando valores
- ▶ Solução:
 - ▶ subconjunto $I \subseteq \{1, 2, \dots, n\}$ tal que $\sum_{i \in I} w_i \leq W$
- ▶ Objetivo:
 - ▶ maximizar $\sum_{i \in I} c_i$

Podemos fazer algumas suposições

1. $\sum_{i=1}^n w_i > W$
2. $0 < w_i \leq W$ para todo $i = 1, \dots, n$

Problema da mochila binário

Problema:

- ▶ Entrada:
 - ▶ inteiro não negativo W representando a capacidade
 - ▶ vetor de inteiros não negativos w representando pesos
 - ▶ vetor de inteiros não negativos c representando valores
- ▶ Solução:
 - ▶ subconjunto $I \subseteq \{1, 2, \dots, n\}$ tal que $\sum_{i \in I} w_i \leq W$
- ▶ Objetivo:
 - ▶ maximizar $\sum_{i \in I} c_i$

Podemos fazer algumas suposições

1. $\sum_{i=1}^n w_i > W$
2. $0 < w_i \leq W$ para todo $i = 1, \dots, n$

Problema da mochila binário

Problema:

- ▶ Entrada:
 - ▶ inteiro não negativo W representando a capacidade
 - ▶ vetor de inteiros não negativos w representando pesos
 - ▶ vetor de inteiros não negativos c representando valores
- ▶ Solução:
 - ▶ subconjunto $I \subseteq \{1, 2, \dots, n\}$ tal que $\sum_{i \in I} w_i \leq W$
- ▶ Objetivo:
 - ▶ maximizar $\sum_{i \in I} c_i$

Podemos fazer algumas suposições

1. $\sum_{i=1}^n w_i > W$
2. $0 < w_i \leq W$ para todo $i = 1, \dots, n$

Problema da mochila binário

Problema:

- ▶ Entrada:
 - ▶ inteiro não negativo W representando a capacidade
 - ▶ vetor de inteiros não negativos w representando pesos
 - ▶ vetor de inteiros não negativos c representando valores
- ▶ Solução:
 - ▶ subconjunto $I \subseteq \{1, 2, \dots, n\}$ tal que $\sum_{i \in I} w_i \leq W$
- ▶ Objetivo:
 - ▶ maximizar $\sum_{i \in I} c_i$

Podemos fazer algumas suposições

1. $\sum_{i=1}^n w_i > W$
2. $0 < w_i \leq W$ para todo $i = 1, \dots, n$

Problema da mochila binário

Problema:

- ▶ Entrada:
 - ▶ inteiro não negativo W representando a capacidade
 - ▶ vetor de inteiros não negativos w representando pesos
 - ▶ vetor de inteiros não negativos c representando valores
- ▶ Solução:
 - ▶ subconjunto $I \subseteq \{1, 2, \dots, n\}$ tal que $\sum_{i \in I} w_i \leq W$
- ▶ Objetivo:
 - ▶ maximizar $\sum_{i \in I} c_i$

Podemos fazer algumas suposições

1. $\sum_{i=1}^n w_i > W$
2. $0 < w_i \leq W$ para todo $i = 1, \dots, n$

Problema da mochila binário

Problema:

- ▶ Entrada:
 - ▶ inteiro não negativo W representando a capacidade
 - ▶ vetor de inteiros não negativos w representando pesos
 - ▶ vetor de inteiros não negativos c representando valores
- ▶ Solução:
 - ▶ subconjunto $I \subseteq \{1, 2, \dots, n\}$ tal que $\sum_{i \in I} w_i \leq W$
- ▶ Objetivo:
 - ▶ maximizar $\sum_{i \in I} c_i$

Podemos fazer algumas suposições

1. $\sum_{i=1}^n w_i > W$
2. $0 < w_i \leq W$ para todo $i = 1, \dots, n$

Problema da mochila binário

Problema:

- ▶ Entrada:
 - ▶ inteiro não negativo W representando a capacidade
 - ▶ vetor de inteiros não negativos w representando pesos
 - ▶ vetor de inteiros não negativos c representando valores
- ▶ Solução:
 - ▶ subconjunto $I \subseteq \{1, 2, \dots, n\}$ tal que $\sum_{i \in I} w_i \leq W$
- ▶ Objetivo:
 - ▶ maximizar $\sum_{i \in I} c_i$

Podemos fazer algumas suposições

1. $\sum_{i=1}^n w_i > W$
2. $0 < w_i \leq W$ para todo $i = 1, \dots, n$

Problema da mochila binário

Problema:

- ▶ Entrada:
 - ▶ inteiro não negativo W representando a capacidade
 - ▶ vetor de inteiros não negativos w representando pesos
 - ▶ vetor de inteiros não negativos c representando valores
- ▶ Solução:
 - ▶ subconjunto $I \subseteq \{1, 2, \dots, n\}$ tal que $\sum_{i \in I} w_i \leq W$
- ▶ Objetivo:
 - ▶ maximizar $\sum_{i \in I} c_i$

Podemos fazer algumas suposições

1. $\sum_{i=1}^n w_i > W$
2. $0 < w_i \leq W$ para todo $i = 1, \dots, n$

Algoritmo de força bruta

- ▶ há 2^n possíveis subconjuntos de itens
- ▶ decidir todos os itens da solução ótima é impraticável!

Podemos decidir se o **último item** está na solução

1. se ele estiver

- ▶ ainda restará capacidade $W - w_n$ na mochila
- ▶ teremos que escolher entre os $n - 1$ primeiros itens

2. se ele **não** estiver

- ▶ ainda restará capacidade W na mochila
- ▶ teremos que escolher entre os $n - 1$ primeiros itens

Algoritmo de força bruta

- ▶ há 2^n possíveis subconjuntos de itens
- ▶ decidir todos os itens da solução ótima é impraticável!

Podemos decidir se o **último item** está na solução

1. se ele estiver
 - ▶ ainda restará capacidade $W - w_n$ na mochila
 - ▶ teremos que escolher entre os $n - 1$ primeiros itens
2. se ele **não** estiver
 - ▶ ainda restará capacidade W na mochila
 - ▶ teremos que escolher entre os $n - 1$ primeiros itens

Algoritmo de força bruta

- ▶ há 2^n possíveis subconjuntos de itens
- ▶ decidir todos os itens da solução ótima é impraticável!

Podemos decidir se o último item está na solução

1. se ele estiver

- ▶ ainda restará capacidade $W - w_n$ na mochila
- ▶ teremos que escolher entre os $n - 1$ primeiros itens

2. se ele não estiver

- ▶ ainda restará capacidade W na mochila
- ▶ teremos que escolher entre os $n - 1$ primeiros itens

Algoritmo de força bruta

- ▶ há 2^n possíveis subconjuntos de itens
- ▶ decidir todos os itens da solução ótima é impraticável!

Podemos decidir se o **último item** está na solução

1. se ele estiver

- ▶ ainda restará capacidade $W - w_n$ na mochila
- ▶ teremos que escolher entre os $n - 1$ primeiros itens

2. se ele **não** estiver

- ▶ ainda restará capacidade W na mochila
- ▶ teremos que escolher entre os $n - 1$ primeiros itens

Algoritmo de força bruta

- ▶ há 2^n possíveis subconjuntos de itens
- ▶ decidir todos os itens da solução ótima é impraticável!

Podemos decidir se o **último item** está na solução

1. se ele estiver

- ▶ ainda restará capacidade $W - w_n$ na mochila
- ▶ teremos que escolher entre os $n - 1$ primeiros itens

2. se ele **não** estiver

- ▶ ainda restará capacidade W na mochila
- ▶ teremos que escolher entre os $n - 1$ primeiros itens

Subestrutura ótima

Algoritmo de força bruta

- ▶ há 2^n possíveis subconjuntos de itens
- ▶ decidir todos os itens da solução ótima é impraticável!

Podemos decidir se o **último item** está na solução

1. se ele estiver

- ▶ ainda restará capacidade $W - w_n$ na mochila
- ▶ teremos que escolher entre os $n - 1$ primeiros itens

2. se ele **não** estiver

- ▶ ainda restará capacidade W na mochila
- ▶ teremos que escolher entre os $n - 1$ primeiros itens

Algoritmo de força bruta

- ▶ há 2^n possíveis subconjuntos de itens
- ▶ decidir todos os itens da solução ótima é impraticável!

Podemos decidir se o **último item** está na solução

1. se ele estiver

- ▶ ainda restará capacidade $W - w_n$ na mochila
- ▶ teremos que escolher entre os $n - 1$ primeiros itens

2. se ele **não** estiver

- ▶ ainda restará capacidade W na mochila
- ▶ teremos que escolher entre os $n - 1$ primeiros itens

Algoritmo de força bruta

- ▶ há 2^n possíveis subconjuntos de itens
- ▶ decidir todos os itens da solução ótima é impraticável!

Podemos decidir se o **último item** está na solução

1. se ele estiver

- ▶ ainda restará capacidade $W - w_n$ na mochila
- ▶ teremos que escolher entre os $n - 1$ primeiros itens

2. se ele **não** estiver

- ▶ ainda restará capacidade W na mochila
- ▶ teremos que escolher entre os $n - 1$ primeiros itens

Algoritmo de força bruta

- ▶ há 2^n possíveis subconjuntos de itens
- ▶ decidir todos os itens da solução ótima é impraticável!

Podemos decidir se o **último item** está na solução

1. se ele estiver

- ▶ ainda restará capacidade $W - w_n$ na mochila
- ▶ teremos que escolher entre os $n - 1$ primeiros itens

2. se ele **não** estiver

- ▶ ainda restará capacidade W na mochila
- ▶ teremos que escolher entre os $n - 1$ primeiros itens

Substrutura ótima

Algoritmo de força bruta

- ▶ há 2^n possíveis subconjuntos de itens
- ▶ decidir todos os itens da solução ótima é impraticável!

Podemos decidir se o **último item** está na solução

1. se ele estiver

- ▶ ainda restará capacidade $W - w_n$ na mochila
- ▶ teremos que escolher entre os $n - 1$ primeiros itens

2. se ele **não** estiver

- ▶ ainda restará capacidade W na mochila
- ▶ teremos que escolher entre os $n - 1$ primeiros itens

Subproblema

Definimos o seguinte subproblema

- ▶ seja d a capacidade residual da mochila
- ▶ considere os k primeiros itens
- ▶ defina $z[k, d]$ o **valor máximo** de um subconjunto $I' \subseteq \{1, 2, \dots, k\}$ tal que $\sum_{i \in I'} w_i \leq d$

Podemos computar $z[k, d]$ com a recorrência:

$$z[k, d] = \begin{cases} 0 & \text{se } k = 0, \text{ ou} \\ z[k - 1, d] & \text{se } w_k > d, \text{ ou} \\ \max\{z[k - 1, d], z[k - 1, d - w_k] + c_k\} & \text{se } w_k \leq d \end{cases}$$

Subproblema

Definimos o seguinte subproblema

- ▶ seja d a capacidade residual da mochila
- ▶ considere os k primeiros itens
- ▶ defina $z[k, d]$ o **valor máximo** de um subconjunto $I' \subseteq \{1, 2, \dots, k\}$ tal que $\sum_{i \in I'} w_i \leq d$

Podemos computar $z[k, d]$ com a recorrência:

$$z[k, d] = \begin{cases} 0 & \text{se } k = 0, \text{ ou} \\ z[k - 1, d] & \text{se } w_k > d, \text{ ou} \\ \max\{z[k - 1, d], z[k - 1, d - w_k] + c_k\} & \text{se } w_k \leq d \end{cases}$$

Subproblema

Definimos o seguinte subproblema

- ▶ seja d a capacidade residual da mochila
- ▶ considere os k primeiros itens
- ▶ defina $z[k, d]$ o **valor máximo** de um subconjunto $I' \subseteq \{1, 2, \dots, k\}$ tal que $\sum_{i \in I'} w_i \leq d$

Podemos computar $z[k, d]$ com a recorrência:

$$z[k, d] = \begin{cases} 0 & \text{se } k = 0, \text{ ou} \\ z[k - 1, d] & \text{se } w_k > d, \text{ ou} \\ \max\{z[k - 1, d], z[k - 1, d - w_k] + c_k\} & \text{se } w_k \leq d \end{cases}$$

Subproblema

Definimos o seguinte subproblema

- ▶ seja d a capacidade residual da mochila
- ▶ considere os k primeiros itens
- ▶ defina $z[k, d]$ o **valor máximo** de um subconjunto $I' \subseteq \{1, 2, \dots, k\}$ tal que $\sum_{i \in I'} w_i \leq d$

Podemos computar $z[k, d]$ com a recorrência:

$$z[k, d] = \begin{cases} 0 & \text{se } k = 0, \text{ ou} \\ z[k - 1, d] & \text{se } w_k > d, \text{ ou} \\ \max\{z[k - 1, d], z[k - 1, d - w_k] + c_k\} & \text{se } w_k \leq d \end{cases}$$

Subproblema

Definimos o seguinte subproblema

- ▶ seja d a capacidade residual da mochila
- ▶ considere os k primeiros itens
- ▶ defina $z[k, d]$ o **valor máximo** de um subconjunto $I' \subseteq \{1, 2, \dots, k\}$ tal que $\sum_{i \in I'} w_i \leq d$

Podemos computar $z[k, d]$ com a recorrência:

$$z[k, d] = \begin{cases} 0 & \text{se } k = 0, \text{ ou} \\ z[k - 1, d] & \text{se } w_k > d, \text{ ou} \\ \max\{z[k - 1, d], z[k - 1, d - w_k] + c_k\} & \text{se } w_k \leq d \end{cases}$$

Subproblema

Definimos o seguinte subproblema

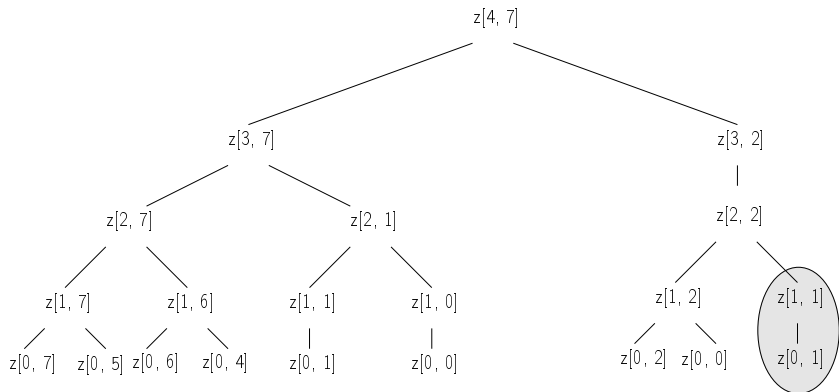
- ▶ seja d a capacidade residual da mochila
- ▶ considere os k primeiros itens
- ▶ defina $z[k, d]$ o **valor máximo** de um subconjunto $I' \subseteq \{1, 2, \dots, k\}$ tal que $\sum_{i \in I'} w_i \leq d$

Podemos computar $z[k, d]$ com a recorrência:

$$z[k, d] = \begin{cases} 0 & \text{se } k = 0, \text{ ou} \\ z[k - 1, d] & \text{se } w_k > d, \text{ ou} \\ \max\{z[k - 1, d], z[k - 1, d - w_k] + c_k\} & \text{se } w_k \leq d \end{cases}$$

Sobreposição de subproblemas

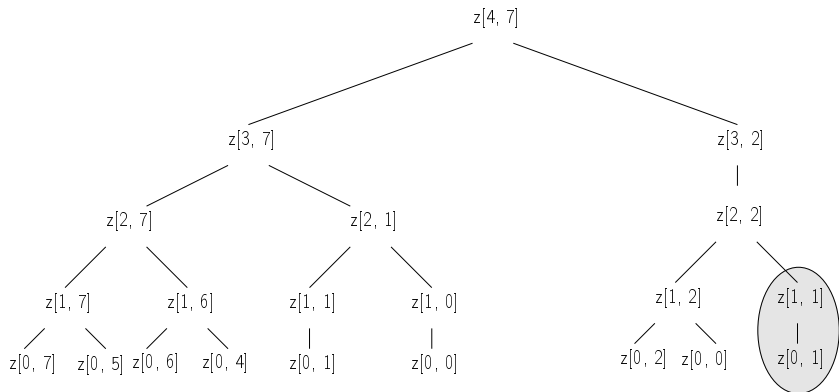
Considere a árvore de recorrência para $W = 7$ e $w = \langle 2, 1, 6, 5 \rangle$:



O subproblema $z[1, 1]$ é computado duas vezes.

Sobreposição de subproblemas

Considere a árvore de recorrência para $W = 7$ e $w = \langle 2, 1, 6, 5 \rangle$:



O subproblema $z[1, 1]$ é computado duas vezes.

Observe que as capacidades são **inteiros**

- ▶ criamos uma tabela com $n + 1$ linhas e $W + 1$ colunas
- ▶ o número de entradas da tabela é $(n + 1)(W + 1)$
- ▶ inicializamos a primeira linha com 0
- ▶ o cálculo de uma entrada só depende da **linha anterior!**
- ▶ o valor do problema original será $z[n, W]$

Observe que as capacidades são **inteiros**

- ▶ criamos uma tabela com $n + 1$ linhas e $W + 1$ colunas
- ▶ o número de entradas da tabela é $(n + 1)(W + 1)$
- ▶ inicializamos a primeira linha com 0
- ▶ o cálculo de uma entrada só depende da **linha anterior!**
- ▶ o valor do problema original será $z[n, W]$

Observe que as capacidades são **inteiros**

- ▶ criamos uma tabela com $n + 1$ linhas e $W + 1$ colunas
- ▶ o número de entradas da tabela é $(n + 1)(W + 1)$
- ▶ inicializamos a primeira linha com 0
- ▶ o cálculo de uma entrada só depende da **linha anterior!**
- ▶ o valor do problema original será $z[n, W]$

Observe que as capacidades são **inteiros**

- ▶ criamos uma tabela com $n + 1$ linhas e $W + 1$ colunas
- ▶ o número de entradas da tabela é $(n + 1)(W + 1)$
- ▶ inicializamos a primeira linha com 0
- ▶ o cálculo de uma entrada só depende da **linha anterior!**
- ▶ o valor do problema original será $z[n, W]$

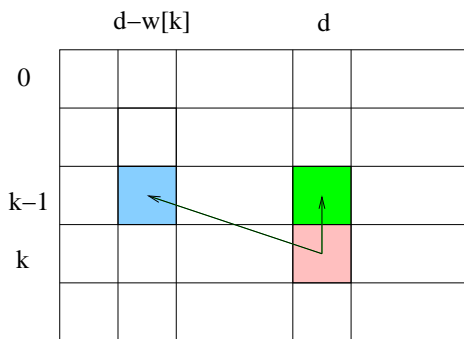
Observe que as capacidades são **inteiros**

- ▶ criamos uma tabela com $n + 1$ linhas e $W + 1$ colunas
- ▶ o número de entradas da tabela é $(n + 1)(W + 1)$
- ▶ inicializamos a primeira linha com 0
- ▶ o cálculo de uma entrada só depende da **linha anterior!**
- ▶ o valor do problema original será $z[n, W]$

Observe que as capacidades são **inteiros**

- ▶ criamos uma tabela com $n + 1$ linhas e $W + 1$ colunas
- ▶ o número de entradas da tabela é $(n + 1)(W + 1)$
- ▶ inicializamos a primeira linha com 0
- ▶ o cálculo de uma entrada só depende da **linha anterior!**
- ▶ o valor do problema original será $z[n, W]$

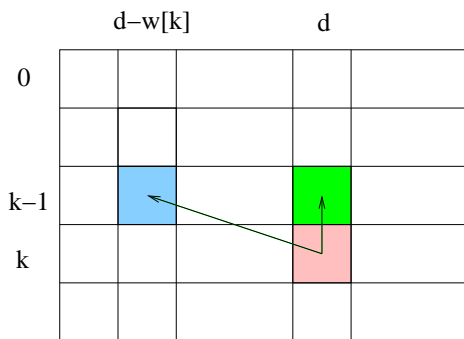
Ordem de preenchimento



$$z[k,d] = \max \{ z[k-1,d], z[k-1,d-w[k]] + c[k] \}$$

O cálculo de $z[k, d]$ depende de $z[k-1, d]$ e $z[k-1, d-w_k]$.

Ordem de preenchimento



$$z[k, d] = \max \{ z[k-1, d], z[k-1, d-w[k]] + c[k] \}$$

O cálculo de $z[k, d]$ depende de $z[k-1, d]$ e $z[k-1, d-w_k]$.

Algoritmo de programação dinâmica

MOCHILA(W, w, c, n)

1 para $d \leftarrow 0$ até W faça

2 $z[0, d] \leftarrow 0$

3 para $k \leftarrow 1$ até n faça

4 para $d \leftarrow 0$ até W faça

5 $z[k, d] \leftarrow z[k - 1, d]$

6 se $w_k \leq d$ e $z[k, d] < c_k + z[k - 1, d - w_k]$ então

7 $z[k, d] \leftarrow c_k + z[k - 1, d - w_k]$

8 devolva $z[n, W]$

Exemplo de preenchimento

Tome $W = 7$, $w = \langle 2, 1, 6, 5 \rangle$ e $c = \langle 10, 7, 25, 24 \rangle$

$k \backslash d$	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1								
2								
3								
4								

Exemplo de preenchimento

Tome $W = 7$, $w = \langle 2, 1, 6, 5 \rangle$ e $c = \langle 10, 7, 25, 24 \rangle$

$k \backslash d$	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2								
3								
4								

Exemplo de preenchimento

Tome $W = 7$, $w = \langle 2, 1, 6, 5 \rangle$ e $c = \langle 10, 7, 25, 24 \rangle$

$k \backslash d$	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3								
4								

Exemplo de preenchimento

Tome $W = 7$, $w = \langle 2, 1, 6, 5 \rangle$ e $c = \langle 10, 7, 25, 24 \rangle$

$k \backslash d$	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4								

Exemplo de preenchimento

Tome $W = 7$, $w = \langle 2, 1, 6, 5 \rangle$ e $c = \langle 10, 7, 25, 24 \rangle$

$k \backslash d$	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Exemplo de preenchimento

Tome $W = 7$, $w = \langle 2, 1, 6, 5 \rangle$ e $c = \langle 10, 7, 25, 24 \rangle$

$k \backslash d$	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Tempo de execução

- ▶ a complexidade de tempo é $O(nW)$
- ▶ é um algoritmo **pseudo-polinomial**
 - ▶ a complexidade é polinomial em n e W
 - ▶ mas W é um número da entrada

Recuperando a solução

- ▶ o algoritmo **não** devolve uma solução ótima
- ▶ podemos obtê-la a partir da tabela z

Tempo de execução

- ▶ a complexidade de tempo é $O(nW)$
- ▶ é um algoritmo **pseudo-polinomial**
 - ▶ a complexidade é polinomial em n e W
 - ▶ mas W é um número da entrada

Recuperando a solução

- ▶ o algoritmo **não** devolve uma solução ótima
- ▶ podemos obtê-la a partir da tabela z

Tempo de execução

- ▶ a complexidade de tempo é $O(nW)$
- ▶ é um algoritmo **pseudo-polinomial**
 - ▶ a complexidade é polinomial em n e W
 - ▶ mas W é um número da entrada

Recuperando a solução

- ▶ o algoritmo **não** devolve uma solução ótima
- ▶ podemos obtê-la a partir da tabela z

Tempo de execução

- ▶ a complexidade de tempo é $O(nW)$
- ▶ é um algoritmo **pseudo-polinomial**
 - ▶ a complexidade é polinomial em n e W
 - ▶ mas W é um número da entrada

Recuperando a solução

- ▶ o algoritmo **não** devolve uma solução ótima
- ▶ podemos obtê-la a partir da tabela z

Tempo de execução

- ▶ a complexidade de tempo é $O(nW)$
- ▶ é um algoritmo **pseudo-polinomial**
 - ▶ a complexidade é polinomial em n e W
 - ▶ mas W é um número da entrada

Recuperando a solução

- ▶ o algoritmo **não** devolve uma solução ótima
- ▶ podemos obtê-la a partir da tabela z

Tempo de execução

- ▶ a complexidade de tempo é $O(nW)$
- ▶ é um algoritmo **pseudo-polinomial**
 - ▶ a complexidade é polinomial em n e W
 - ▶ mas W é um número da entrada

Recuperando a solução

- ▶ o algoritmo **não** devolve uma solução ótima
- ▶ podemos obtê-la a partir da tabela z

Tempo de execução

- ▶ a complexidade de tempo é $O(nW)$
- ▶ é um algoritmo **pseudo-polinomial**
 - ▶ a complexidade é polinomial em n e W
 - ▶ mas W é um número da entrada

Recuperando a solução

- ▶ o algoritmo **não** devolve uma solução ótima
- ▶ podemos obtê-la a partir da tabela z

Tempo de execução

- ▶ a complexidade de tempo é $O(nW)$
- ▶ é um algoritmo **pseudo-polinomial**
 - ▶ a complexidade é polinomial em n e W
 - ▶ mas W é um número da entrada

Recuperando a solução

- ▶ o algoritmo **não** devolve uma solução ótima
- ▶ podemos obtê-la a partir da tabela z

Recuperação da solução

MOCHILA-SOLUCAO(W, z, n)

```
1   $d \leftarrow W$ 
2  para  $k \leftarrow n$  decrescendo até 1 faça
3      se  $z[k, d] = z[k - 1, d]$  então
4           $x[k] \leftarrow 0$ 
5      senão
6           $x[k] \leftarrow 1$ 
7           $d \leftarrow d - w_k$ 
8  devolva  $x$ 
```

- ▶ o vetor x que indica os itens de uma solução ótima

Recuperação da solução

MOCHILA-SOLUCAO(W, z, n)

```
1   $d \leftarrow W$ 
2  para  $k \leftarrow n$  decrescendo até 1 faça
3      se  $z[k, d] = z[k - 1, d]$  então
4           $x[k] \leftarrow 0$ 
5      senão
6           $x[k] \leftarrow 1$ 
7           $d \leftarrow d - w_k$ 
8  devolva  $x$ 
```

- ▶ o vetor x que indica os itens de uma solução ótima

Exemplo de recuperação

Tome $W = 7$, $w = \langle 2, 1, 6, 5 \rangle$ e $c = \langle 10, 7, 25, 24 \rangle$

$k \backslash d$	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Exemplo de recuperação

Tome $W = 7$, $w = \langle 2, 1, 6, 5 \rangle$ e $c = \langle 10, 7, 25, 24 \rangle$

$k \backslash d$	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Exemplo de recuperação

Tome $W = 7$, $w = \langle 2, 1, 6, 5 \rangle$ e $c = \langle 10, 7, 25, 24 \rangle$

$k \backslash d$	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Exemplo de recuperação

Tome $W = 7$, $w = \langle 2, 1, 6, 5 \rangle$ e $c = \langle 10, 7, 25, 24 \rangle$

$k \backslash d$	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Exemplo de recuperação

Tome $W = 7$, $w = \langle 2, 1, 6, 5 \rangle$ e $c = \langle 10, 7, 25, 24 \rangle$

$k \backslash d$	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

$$x[1] = x[4] = 1, \quad x[2] = x[3] = 0$$

É possível economizar memória

- ▶ a complexidade de espaço é $O(nW)$
- ▶ precisamos apenas da última linha para a recorrência
- ▶ em dado momento, mantemos no máximo **duas linhas**
- ▶ o algoritmo melhorado usa apenas $O(W)$ de espaço
- ▶ mas isso inviabiliza a recuperação da solução

Complexidade de espaço

É possível economizar memória

- ▶ a complexidade de espaço é $O(nW)$
- ▶ precisamos apenas da última linha para a recorrência
- ▶ em dado momento, mantemos no máximo **duas linhas**
- ▶ o algoritmo melhorado usa apenas $O(W)$ de espaço
- ▶ mas isso inviabiliza a recuperação da solução

Complexidade de espaço

É possível economizar memória

- ▶ a complexidade de espaço é $O(nW)$
- ▶ precisamos apenas da última linha para a recorrência
- ▶ em dado momento, mantemos no máximo duas linhas
- ▶ o algoritmo melhorado usa apenas $O(W)$ de espaço
- ▶ mas isso inviabiliza a recuperação da solução

Complexidade de espaço

É possível economizar memória

- ▶ a complexidade de espaço é $O(nW)$
- ▶ precisamos apenas da última linha para a recorrência
- ▶ em dado momento, mantemos no máximo **duas linhas**
- ▶ o algoritmo melhorado usa apenas $O(W)$ de espaço
- ▶ mas isso inviabiliza a recuperação da solução

Complexidade de espaço

É possível economizar memória

- ▶ a complexidade de espaço é $O(nW)$
- ▶ precisamos apenas da última linha para a recorrência
- ▶ em dado momento, mantemos no máximo **duas linhas**
- ▶ o algoritmo melhorado usa apenas $O(W)$ de espaço
- ▶ mas isso inviabiliza a recuperação da solução

Complexidade de espaço

É possível economizar memória

- ▶ a complexidade de espaço é $O(nW)$
- ▶ precisamos apenas da última linha para a recorrência
- ▶ em dado momento, mantemos no máximo **duas linhas**
- ▶ o algoritmo melhorado usa apenas $O(W)$ de espaço
- ▶ mas isso inviabiliza a recuperação da solução

Programação dinâmica

- ▶ Problema da árvore de busca ótima

Árvores binárias de busca

Considere um conjunto de elementos

- ▶ elas podem ser ordenadas $e_1 < e_2 < \dots < e_n$
- ▶ e a frequência de consulta f_i para cada chave e_i

Vamos criar uma árvore binária de busca

- ▶ respeita a **propriedade de busca**
- ▶ consultar uma chave acessa os nós antecedentes

Pergunta: qual árvore minimiza o número de acessos a nós?

Árvores binárias de busca

Considere um conjunto de elementos

- ▶ elas podem ser ordenadas $e_1 < e_2 < \dots < e_n$
- ▶ e a frequência de consulta f_i para cada chave e_i

Vamos criar uma árvore binária de busca

- ▶ respeita a **propriedade de busca**
- ▶ consultar uma chave acessa os nós antecedentes

Pergunta: qual árvore minimiza o número de acessos a nós?

Árvores binárias de busca

Considere um conjunto de elementos

- ▶ elas podem ser ordenadas $e_1 < e_2 < \dots < e_n$
- ▶ e a frequência de consulta f_i para cada chave e_i

Vamos criar uma árvore binária de busca

- ▶ respeita a **propriedade de busca**
- ▶ consultar uma chave acessa os nós antecedentes

Pergunta: qual árvore minimiza o número de acessos a nós?

Árvores binárias de busca

Considere um conjunto de elementos

- ▶ elas podem ser ordenadas $e_1 < e_2 < \dots < e_n$
- ▶ e a frequência de consulta f_i para cada chave e_i

Vamos criar uma árvore binária de busca

- ▶ respeita a **propriedade de busca**
- ▶ consultar uma chave acessa os nós antecedentes

Pergunta: qual árvore minimiza o número de acessos a nós?

Árvores binárias de busca

Considere um conjunto de elementos

- ▶ elas podem ser ordenadas $e_1 < e_2 < \dots < e_n$
- ▶ e a frequência de consulta f_i para cada chave e_i

Vamos criar uma árvore binária de busca

- ▶ respeita a **propriedade de busca**
- ▶ consultar uma chave acessa os nós antecedentes

Pergunta: qual árvore minimiza o número de acessos a nós?

Árvores binárias de busca

Considere um conjunto de elementos

- ▶ elas podem ser ordenadas $e_1 < e_2 < \dots < e_n$
- ▶ e a frequência de consulta f_i para cada chave e_i

Vamos criar uma árvore binária de busca

- ▶ respeita a **propriedade de busca**
- ▶ consultar uma chave acessa os nós antecedentes

Pergunta: qual árvore minimiza o número de acessos a nós?

Árvores binárias de busca

Considere um conjunto de elementos

- ▶ elas podem ser ordenadas $e_1 < e_2 < \dots < e_n$
- ▶ e a frequência de consulta f_i para cada chave e_i

Vamos criar uma árvore binária de busca

- ▶ respeita a **propriedade de busca**
- ▶ consultar uma chave acessa os nós antecedentes

Pergunta: qual árvore minimiza o número de acessos a nós?

Exemplo de árvore de busca

Considere quatro chaves:

- ▶ ordenadas como $A \leq B \leq C \leq D$
- ▶ frequências $f_A = 45$, $f_B = 25$, $f_C = 18$ e $f_D = 12$

Exemplo de árvore de busca:

- ▶ o total de nós acessados nesta árvore é 207
- ▶ podemos encontrar uma árvore melhor?

Exemplo de árvore de busca

Considere quatro chaves:

- ▶ ordenadas como $A \leq B \leq C \leq D$
- ▶ frequências $f_A = 45$, $f_B = 25$, $f_C = 18$ e $f_D = 12$

Exemplo de árvore de busca:

- ▶ o total de nós acessados nesta árvore é 207
- ▶ podemos encontrar uma árvore melhor?

Exemplo de árvore de busca

Considere quatro chaves:

- ▶ ordenadas como $A \leq B \leq C \leq D$
- ▶ frequências $f_A = 45$, $f_B = 25$, $f_C = 18$ e $f_D = 12$

Exemplo de árvore de busca:

- ▶ o total de nós acessados nesta árvore é 207
- ▶ podemos encontrar uma árvore melhor?

Exemplo de árvore de busca

Considere quatro chaves:

- ▶ ordenadas como $A \leq B \leq C \leq D$
- ▶ frequências $f_A = 45$, $f_B = 25$, $f_C = 18$ e $f_D = 12$

Exemplo de árvore de busca:

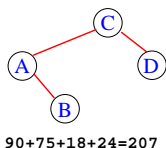
- ▶ o total de nós acessados nesta árvore é 207
- ▶ podemos encontrar uma árvore melhor?

Exemplo de árvore de busca

Considere quatro chaves:

- ▶ ordenadas como $A \leq B \leq C \leq D$
- ▶ frequências $f_A = 45$, $f_B = 25$, $f_C = 18$ e $f_D = 12$

Exemplo de árvore de busca:



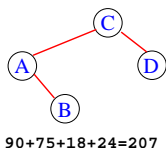
- ▶ o total de nós acessados nesta árvore é 207
- ▶ podemos encontrar uma árvore melhor?

Exemplo de árvore de busca

Considere quatro chaves:

- ▶ ordenadas como $A \leq B \leq C \leq D$
- ▶ frequências $f_A = 45$, $f_B = 25$, $f_C = 18$ e $f_D = 12$

Exemplo de árvore de busca:



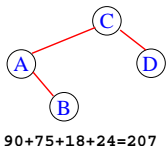
- ▶ o total de nós acessados nesta árvore é 207
- ▶ podemos encontrar uma árvore melhor?

Exemplo de árvore de busca

Considere quatro chaves:

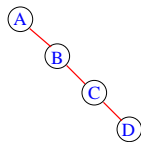
- ▶ ordenadas como $A \leq B \leq C \leq D$
- ▶ frequências $f_A = 45$, $f_B = 25$, $f_C = 18$ e $f_D = 12$

Exemplo de árvore de busca:

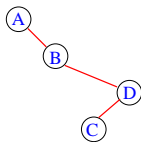


- ▶ o total de nós acessados nesta árvore é 207
- ▶ podemos encontrar uma árvore melhor?

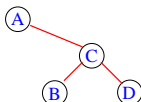
Listando todas as árvores de busca



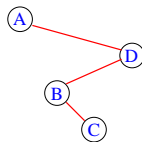
$$45+50+54+48=197$$



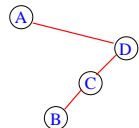
$$45+50+72+36=203$$



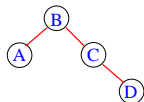
$$45+75+36+36=192$$



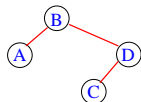
$$45+75+72+24=216$$



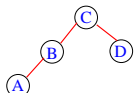
$$45+100+54+24=223$$



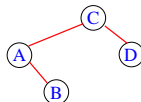
$$90+25+36+36=187$$



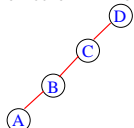
$$90+25+54+24=193$$



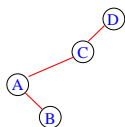
$$135+50+18+24=227$$



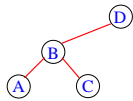
$$90+75+18+24=207$$



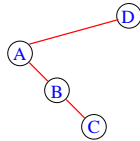
$$180+75+36+12=303$$



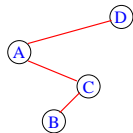
$$135+100+36+12=283$$



$$135+50+54+12=251$$



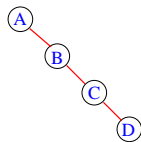
$$90+75+72+12=249$$



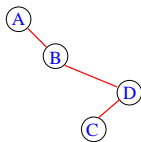
$$90+100+54+12=256$$

- ▶ é **impraticável** listar todas as árvores!
- ▶ exercício: quantas árvores de busca com n nós existem?

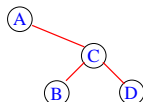
Listando todas as árvores de busca



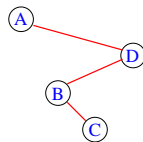
$$45+50+54+48=197$$



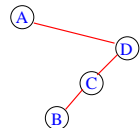
$$45+50+72+36=203$$



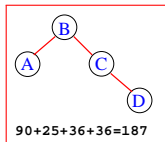
$$45+75+36+36=192$$



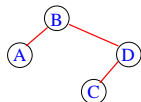
$$45+75+72+24=216$$



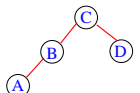
$$45+100+54+24=223$$



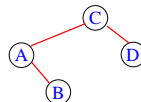
$$90+25+36+36=187$$



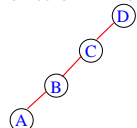
$$90+25+54+24=193$$



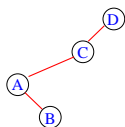
$$135+50+18+24=227$$



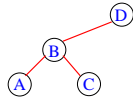
$$90+75+18+24=207$$



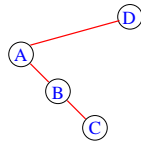
$$180+75+36+12=303$$



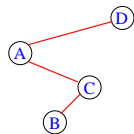
$$135+100+36+12=283$$



$$135+50+54+12=251$$



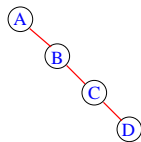
$$90+75+72+12=249$$



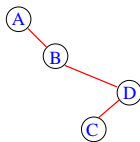
$$90+100+54+12=256$$

- ▶ é **impraticável** listar todas as árvores!
- ▶ exercício: quantas árvores de busca com n nós existem?

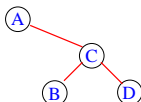
Listando todas as árvores de busca



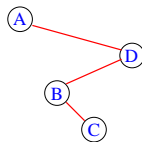
$$45+50+54+48=197$$



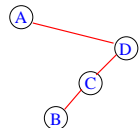
$$45+50+72+36=203$$



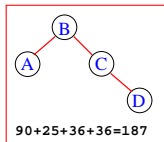
$$45+75+36+36=192$$



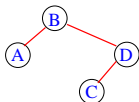
$$45+75+72+24=216$$



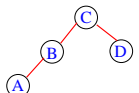
$$45+100+54+24=223$$



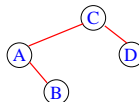
$$90+25+36+36=187$$



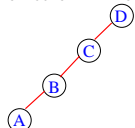
$$90+25+54+24=193$$



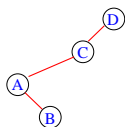
$$135+50+18+24=227$$



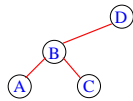
$$90+75+18+24=207$$



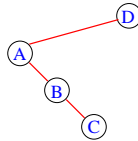
$$180+75+36+12=303$$



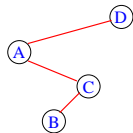
$$135+100+36+12=283$$



$$135+50+54+12=251$$



$$90+75+72+12=249$$

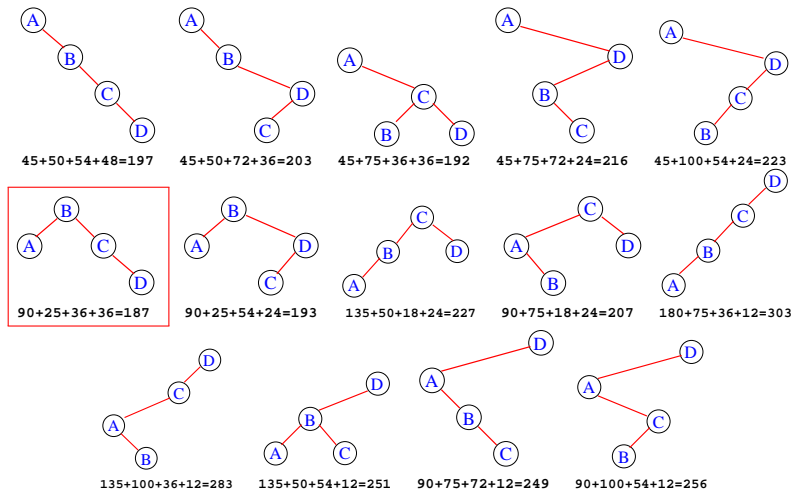


$$90+100+54+12=256$$

▶ é **impraticável** listar todas as árvores!

▶ exercício: quantas árvores de busca com n nós existem?

Listando todas as árvores de busca



- ▶ é **impraticável** listar todas as árvores!
- ▶ exercício: quantas árvores de busca com n nós existem?

Problema da árvore de busca ótima

Problema:

- ▶ Entrada:
 - ▶ sequência de chaves $e_1 < e_2 < \dots < e_n$
 - ▶ frequências de consulta f_i para cada chave e_i
- ▶ Solução:
 - ▶ árvore binária de busca com nós e_1, e_2, \dots, e_n
- ▶ Objetivo:
 - ▶ minimizar número total de nós consultados

Problema da árvore de busca ótima

Problema:

- ▶ Entrada:
 - ▶ sequência de chaves $e_1 < e_2 < \dots < e_n$
 - ▶ frequências de consulta f_i para cada chave e_i
- ▶ Solução:
 - ▶ árvore binária de busca com nós e_1, e_2, \dots, e_n
- ▶ Objetivo:
 - ▶ minimizar número total de nós consultados

Problema da árvore de busca ótima

Problema:

- ▶ Entrada:
 - ▶ sequência de chaves $e_1 < e_2 < \dots < e_n$
 - ▶ frequências de consulta f_i para cada chave e_i
- ▶ Solução:
 - ▶ árvore binária de busca com nós e_1, e_2, \dots, e_n
- ▶ Objetivo:
 - ▶ minimizar número total de nós consultados

Problema da árvore de busca ótima

Problema:

- ▶ Entrada:
 - ▶ sequência de chaves $e_1 < e_2 < \dots < e_n$
 - ▶ frequências de consulta f_i para cada chave e_i
- ▶ Solução:
 - ▶ árvore binária de busca com nós e_1, e_2, \dots, e_n
- ▶ Objetivo:
 - ▶ minimizar número total de nós consultados

Problema da árvore de busca ótima

Problema:

- ▶ Entrada:
 - ▶ sequência de chaves $e_1 < e_2 < \dots < e_n$
 - ▶ frequências de consulta f_i para cada chave e_i
- ▶ Solução:
 - ▶ árvore binária de busca com nós e_1, e_2, \dots, e_n
- ▶ Objetivo:
 - ▶ minimizar número total de nós consultados

Problema da árvore de busca ótima

Problema:

- ▶ Entrada:
 - ▶ sequência de chaves $e_1 < e_2 < \dots < e_n$
 - ▶ frequências de consulta f_i para cada chave e_i
- ▶ Solução:
 - ▶ árvore binária de busca com nós e_1, e_2, \dots, e_n
- ▶ Objetivo:
 - ▶ minimizar número total de nós consultados

Problema da árvore de busca ótima

Problema:

- ▶ Entrada:
 - ▶ sequência de chaves $e_1 < e_2 < \dots < e_n$
 - ▶ frequências de consulta f_i para cada chave e_i
- ▶ Solução:
 - ▶ árvore binária de busca com nós e_1, e_2, \dots, e_n
- ▶ Objetivo:
 - ▶ **minimizar** número total de nós consultados

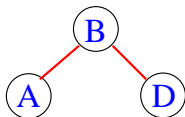
Problema da árvore de busca ótima

Problema:

- ▶ Entrada:
 - ▶ sequência de chaves $e_1 < e_2 < \dots < e_n$
 - ▶ frequências de consulta f_i para cada chave e_i
- ▶ Solução:
 - ▶ árvore binária de busca com nós e_1, e_2, \dots, e_n
- ▶ Objetivo:
 - ▶ **minimizar** número total de nós consultados

Propriedade da árvore de busca

Considere a árvore a seguir



Seja T uma árvore de busca com $A \leq B \leq C \leq D$

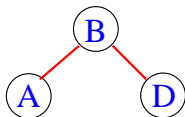
- ▶ pergunta: a árvore acima pode ser subárvore de T ?
- ▶ resposta: **não**, ela deveria conter o elemento C

Denote por $T(v)$ a subárvore enraizada em v

- ▶ suponha que $T(v)$ contém e_i e e_k
- ▶ então $T(v)$ contém **todos** elementos de e_i a e_k

Propriedade da árvore de busca

Considere a árvore a seguir



Seja T uma árvore de busca com $A \leq B \leq C \leq D$

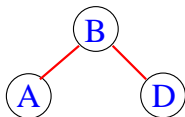
- ▶ pergunta: a árvore acima pode ser subárvore de T ?
- ▶ resposta: **não**, ela deveria conter o elemento C

Denote por $T(v)$ a subárvore enraizada em v

- ▶ suponha que $T(v)$ contém e_i e e_k
- ▶ então $T(v)$ contém **todos** elementos de e_i a e_k

Propriedade da árvore de busca

Considere a árvore a seguir



Seja T uma árvore de busca com $A \leq B \leq C \leq D$

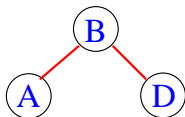
- ▶ **pergunta:** a árvore acima pode ser subárvore de T ?
- ▶ **resposta:** não, ela deveria conter o elemento C

Denote por $T(v)$ a subárvore enraizada em v

- ▶ suponha que $T(v)$ contém e_i e e_k
- ▶ então $T(v)$ contém **todos** elementos de e_i a e_k

Propriedade da árvore de busca

Considere a árvore a seguir



Seja T uma árvore de busca com $A \leq B \leq C \leq D$

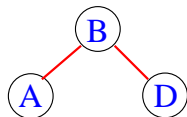
- ▶ **pergunta:** a árvore acima pode ser subárvore de T ?
- ▶ **resposta:** **não**, ela deveria conter o elemento C

Denote por $T(v)$ a subárvore enraizada em v

- ▶ suponha que $T(v)$ contém e_i e e_k
- ▶ então $T(v)$ contém **todos** elementos de e_i a e_k

Propriedade da árvore de busca

Considere a árvore a seguir



Seja T uma árvore de busca com $A \leq B \leq C \leq D$

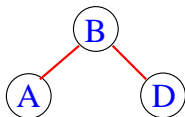
- ▶ **pergunta:** a árvore acima pode ser subárvore de T ?
- ▶ **resposta:** não, ela deveria conter o elemento C

Denote por $T(v)$ a subárvore enraizada em v

- ▶ suponha que $T(v)$ contém e_i e e_k
- ▶ então $T(v)$ contém todos elementos de e_i a e_k

Propriedade da árvore de busca

Considere a árvore a seguir



Seja T uma árvore de busca com $A \leq B \leq C \leq D$

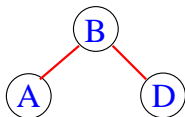
- ▶ **pergunta:** a árvore acima pode ser subárvore de T ?
- ▶ **resposta:** não, ela deveria conter o elemento C

Denote por $T(v)$ a subárvore enraizada em v

- ▶ suponha que $T(v)$ contém e_i e e_k
- ▶ então $T(v)$ contém todos elementos de e_i a e_k

Propriedade da árvore de busca

Considere a árvore a seguir



Seja T uma árvore de busca com $A \leq B \leq C \leq D$

- ▶ **pergunta:** a árvore acima pode ser subárvore de T ?
- ▶ **resposta:** não, ela deveria conter o elemento C

Denote por $T(v)$ a subárvore enraizada em v

- ▶ suponha que $T(v)$ contém e_i e e_k
- ▶ então $T(v)$ contém **todos** elementos de e_i a e_k

Subestrutura ótima

Considere uma árvore de busca T com chave e_j

- ▶ se $T(e_j) = \{e_i, \dots, e_{j-1}, e_j, e_{j+1}, \dots, e_k\}$, então
 - ▶ no ramo esquerdo deve haver os elementos e_i, \dots, e_{j-1} .
 - ▶ no ramo direito deve haver os elementos e_{j+1}, \dots, e_k .

- ▶ **subárvores** de e_i, \dots, e_{j-1} e e_{j+1}, \dots, e_k devem ser ótimas

Decompondo em subproblema

- ▶ qualquer um de e_i, \dots, e_k pode ser a ser raiz
- ▶ o número de acessos à raiz é a soma de todas frequências

Subestrutura ótima

Considere uma árvore de busca T com chave e_j

- ▶ se $T(e_j) = \{e_i, \dots, e_{j-1}, e_j, e_{j+1}, \dots, e_k\}$, então
 - ▶ no ramo esquerdo deve haver os elementos e_i, \dots, e_{j-1} .
 - ▶ no ramo direito deve haver os elementos e_{j+1}, \dots, e_k .

- ▶ **subárvores** de e_i, \dots, e_{j-1} e e_{j+1}, \dots, e_k devem ser ótimas

Decompondo em subproblema

- ▶ qualquer um de e_i, \dots, e_k pode ser a ser raiz
- ▶ o número de acessos à raiz é a soma de todas frequências

Subestrutura ótima

Considere uma árvore de busca T com chave e_j

- ▶ se $T(e_j) = \{e_i, \dots, e_{j-1}, e_j, e_{j+1}, \dots, e_k\}$, então
 - ▶ no ramo esquerdo deve haver os elementos e_i, \dots, e_{j-1} .
 - ▶ no ramo direito deve haver os elementos e_{j+1}, \dots, e_k .

- ▶ **subárvores** de e_i, \dots, e_{j-1} e e_{j+1}, \dots, e_k devem ser ótimas

Decompondo em subproblema

- ▶ qualquer um de e_i, \dots, e_k pode ser a ser raiz
- ▶ o número de acessos à raiz é a soma de todas frequências

Subestrutura ótima

Considere uma árvore de busca T com chave e_j

- ▶ se $T(e_j) = \{e_i, \dots, e_{j-1}, e_j, e_{j+1}, \dots, e_k\}$, então
 - ▶ no ramo esquerdo deve haver os elementos e_i, \dots, e_{j-1} .
 - ▶ no ramo direito deve haver os elementos e_{j+1}, \dots, e_k .

- ▶ **subárvores** de e_i, \dots, e_{j-1} e e_{j+1}, \dots, e_k devem ser ótimas

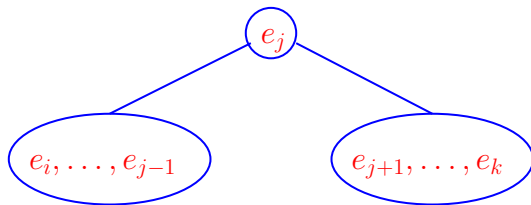
Decompondo em subproblema

- ▶ qualquer um de e_i, \dots, e_k pode ser a ser raiz
- ▶ o número de acessos à raiz é a soma de todas frequências

Subestrutura ótima

Considere uma árvore de busca T com chave e_j

- ▶ se $T(e_j) = \{e_i, \dots, e_{j-1}, e_j, e_{j+1}, \dots, e_k\}$, então
 - ▶ no ramo esquerdo deve haver os elementos e_i, \dots, e_{j-1} .
 - ▶ no ramo direito deve haver os elementos e_{j+1}, \dots, e_k .



- ▶ **subárvores** de e_i, \dots, e_{j-1} e e_{j+1}, \dots, e_k devem ser ótimas

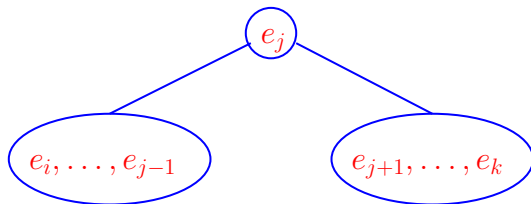
Decompondo em subproblema

- ▶ qualquer um de e_i, \dots, e_k pode ser a ser raiz
- ▶ o número de acessos à raiz é a soma de todas frequências

Subestrutura ótima

Considere uma árvore de busca T com chave e_j

- ▶ se $T(e_j) = \{e_i, \dots, e_{j-1}, e_j, e_{j+1}, \dots, e_k\}$, então
 - ▶ no ramo esquerdo deve haver os elementos e_i, \dots, e_{j-1} .
 - ▶ no ramo direito deve haver os elementos e_{j+1}, \dots, e_k .



- ▶ **subárvores** de e_i, \dots, e_{j-1} e e_{j+1}, \dots, e_k devem ser ótimas

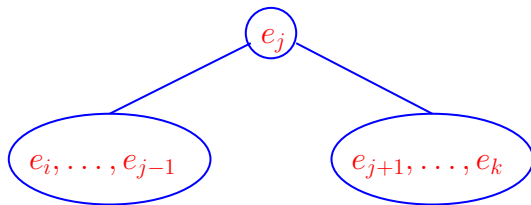
Decompondo em subproblema

- ▶ qualquer um de e_i, \dots, e_k pode ser a ser raiz
- ▶ o número de acessos à raiz é a soma de todas frequências

Subestrutura ótima

Considere uma árvore de busca T com chave e_j

- ▶ se $T(e_j) = \{e_i, \dots, e_{j-1}, e_j, e_{j+1}, \dots, e_k\}$, então
 - ▶ no ramo esquerdo deve haver os elementos e_i, \dots, e_{j-1} .
 - ▶ no ramo direito deve haver os elementos e_{j+1}, \dots, e_k .



- ▶ **subárvores** de e_i, \dots, e_{j-1} e e_{j+1}, \dots, e_k devem ser ótimas

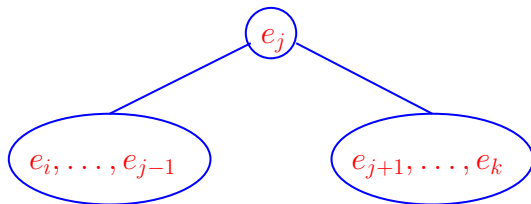
Decompondo em subproblema

- ▶ qualquer um de e_i, \dots, e_k pode ser a ser raiz
- ▶ o número de acessos à raiz é a soma de todas frequências

Subestrutura ótima

Considere uma árvore de busca T com chave e_j

- ▶ se $T(e_j) = \{e_i, \dots, e_{j-1}, e_j, e_{j+1}, \dots, e_k\}$, então
 - ▶ no ramo esquerdo deve haver os elementos e_i, \dots, e_{j-1} .
 - ▶ no ramo direito deve haver os elementos e_{j+1}, \dots, e_k .



- ▶ **subárvores** de e_i, \dots, e_{j-1} e e_{j+1}, \dots, e_k devem ser ótimas

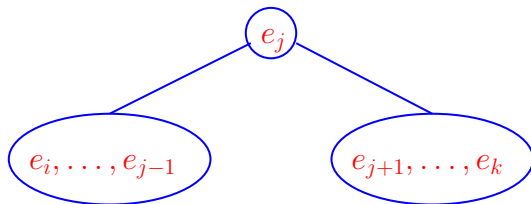
Decompondo em subproblema

- ▶ qualquer um de e_i, \dots, e_k pode ser a ser raiz
- ▶ o número de acessos à raiz é a soma de todas frequências

Subestrutura ótima

Considere uma árvore de busca T com chave e_j

- ▶ se $T(e_j) = \{e_i, \dots, e_{j-1}, e_j, e_{j+1}, \dots, e_k\}$, então
 - ▶ no ramo esquerdo deve haver os elementos e_i, \dots, e_{j-1} .
 - ▶ no ramo direito deve haver os elementos e_{j+1}, \dots, e_k .



- ▶ **subárvores** de e_i, \dots, e_{j-1} e e_{j+1}, \dots, e_k devem ser ótimas

Decompondo em subproblema

- ▶ qualquer um de e_i, \dots, e_k pode ser a ser raiz
- ▶ o número de acessos à raiz é a soma de todas frequências

Subproblema

Definimos o seguinte subproblema

- ▶ considere um par de índices (i, k)
- ▶ seja $a[i, k]$ o **menor** número de acessos em árvore de busca contendo e_i, \dots, e_k

Podemos utilizar a recorrência:

$$a[i, k] = \begin{cases} 0 & \text{se } k < i, \\ \sum_{t=i}^k f_t + \min_{i \leq j \leq k} \left\{ a[i, j-i] + a[j+1, k] \right\} & \text{se } k \geq i \end{cases}$$

Subproblema

Definimos o seguinte subproblema

- ▶ considere um par de índices (i, k)
- ▶ seja $a[i, k]$ o menor número de acessos em árvore de busca contendo e_i, \dots, e_k

Podemos utilizar a recorrência:

$$a[i, k] = \begin{cases} 0 & \text{se } k < i, \\ \sum_{t=i}^k f_t + \min_{i \leq j \leq k} \left\{ a[i, j-i] + a[j+1, k] \right\} & \text{se } k \geq i \end{cases}$$

Subproblema

Definimos o seguinte subproblema

- ▶ considere um par de índices (i, k)
- ▶ seja $a[i, k]$ o **menor** número de acessos em árvore de busca contendo e_i, \dots, e_k

Podemos utilizar a recorrência:

$$a[i, k] = \begin{cases} 0 & \text{se } k < i, \\ \sum_{t=i}^k f_t + \min_{i \leq j \leq k} \left\{ a[i, j-i] + a[j+1, k] \right\} & \text{se } k \geq i \end{cases}$$

Subproblema

Definimos o seguinte subproblema

- ▶ considere um par de índices (i, k)
- ▶ seja $a[i, k]$ o **menor** número de acessos em árvore de busca contendo e_i, \dots, e_k

Podemos utilizar a recorrência:

$$a[i, k] = \begin{cases} 0 & \text{se } k < i, \\ \sum_{t=i}^k f_t + \min_{i \leq j \leq k} \left\{ a[i, j-i] + a[j+1, k] \right\} & \text{se } k \geq i \end{cases}$$

Subproblema

Definimos o seguinte subproblema

- ▶ considere um par de índices (i, k)
- ▶ seja $a[i, k]$ o **menor** número de acessos em árvore de busca contendo e_i, \dots, e_k

Podemos utilizar a recorrência:

$$a[i, k] = \begin{cases} 0 & \text{se } k < i, \\ \sum_{t=i}^k f_t + \min_{i \leq j \leq k} \left\{ a[i, j-i] + a[j+1, k] \right\} & \text{se } k \geq i \end{cases}$$

Algoritmo

- ▶ considere elementos dummy e_0 e e_{n+1} com $f_0 = f_{n+1} = 0$
- ▶ denotamos por t o tamanho da subsequência

ÁRVORE-BUSCA(e, f)

```
1  para  $i \leftarrow 1$  até  $n + 1$  faça
2     $a[i, i - 1] \leftarrow 0$ 
3  para  $t \leftarrow 0$  até  $n - 1$  faça
4    para  $i \leftarrow 1$  até  $n - t$  faça
5       $k \leftarrow i + t$ 
6       $a[i, k] \leftarrow \sum_{j=i}^k f_j + \min_{i \leq j \leq k} \{a[i, j - 1] + a[j + 1, k]\}$ 
7  devolva  $a[1, n]$ 
```

Observações

- ▶ a complexidade de tempo é $O(n^3)$
- ▶ exercício: modifique para encontrar uma solução ótima

Algoritmo

- ▶ considere elementos dummy e_0 e e_{n+1} com $f_0 = f_{n+1} = 0$
- ▶ denotamos por t o tamanho da subsequência

ÁRVORE-BUSCA(e, f)

```
1  para  $i \leftarrow 1$  até  $n + 1$  faça
2     $a[i, i - 1] \leftarrow 0$ 
3  para  $t \leftarrow 0$  até  $n - 1$  faça
4    para  $i \leftarrow 1$  até  $n - t$  faça
5       $k \leftarrow i + t$ 
6       $a[i, k] \leftarrow \sum_{j=i}^k f_j + \min_{i \leq j \leq k} \{a[i, j - 1] + a[j + 1, k]\}$ 
7  devolva  $a[1, n]$ 
```

Observações

- ▶ a complexidade de tempo é $O(n^3)$
- ▶ exercício: modifique para encontrar uma solução ótima

Algoritmo

- ▶ considere elementos dummy e_0 e e_{n+1} com $f_0 = f_{n+1} = 0$
- ▶ denotamos por t o tamanho da subsequência

ÁRVORE-BUSCA(e, f)

```
1  para  $i \leftarrow 1$  até  $n + 1$  faça
2     $a[i, i - 1] \leftarrow 0$ 
3  para  $t \leftarrow 0$  até  $n - 1$  faça
4    para  $i \leftarrow 1$  até  $n - t$  faça
5       $k \leftarrow i + t$ 
6       $a[i, k] \leftarrow \sum_{j=i}^k f_j + \min_{i \leq j \leq k} \{a[i, j - 1] + a[j + 1, k]\}$ 
7  devolva  $a[1, n]$ 
```

Observações

- ▶ a complexidade de tempo é $O(n^3)$
- ▶ exercício: modifique para encontrar uma solução ótima

Algoritmo

- ▶ considere elementos dummy e_0 e e_{n+1} com $f_0 = f_{n+1} = 0$
- ▶ denotamos por t o tamanho da subsequência

ÁRVORE-BUSCA(e, f)

```
1  para  $i \leftarrow 1$  até  $n + 1$  faça
2     $a[i, i - 1] \leftarrow 0$ 
3  para  $t \leftarrow 0$  até  $n - 1$  faça
4    para  $i \leftarrow 1$  até  $n - t$  faça
5       $k \leftarrow i + t$ 
6       $a[i, k] \leftarrow \sum_{j=i}^k f_j + \min_{i \leq j \leq k} \{a[i, j - 1] + a[j + 1, k]\}$ 
7  devolva  $a[1, n]$ 
```

Observações

- ▶ a complexidade de tempo é $O(n^3)$
- ▶ exercício: modifique para encontrar uma solução ótima

Algoritmo

- ▶ considere elementos dummy e_0 e e_{n+1} com $f_0 = f_{n+1} = 0$
- ▶ denotamos por t o tamanho da subsequência

ÁRVORE-BUSCA(e, f)

```
1  para  $i \leftarrow 1$  até  $n + 1$  faça
2     $a[i, i - 1] \leftarrow 0$ 
3  para  $t \leftarrow 0$  até  $n - 1$  faça
4    para  $i \leftarrow 1$  até  $n - t$  faça
5       $k \leftarrow i + t$ 
6       $a[i, k] \leftarrow \sum_{j=i}^k f_j + \min_{i \leq j \leq k} \{a[i, j - 1] + a[j + 1, k]\}$ 
7  devolva  $a[1, n]$ 
```

Observações

- ▶ a complexidade de tempo é $O(n^3)$
- ▶ exercício: modifique para encontrar uma solução ótima

Algoritmo

- ▶ considere elementos dummy e_0 e e_{n+1} com $f_0 = f_{n+1} = 0$
- ▶ denotamos por t o tamanho da subsequência

ÁRVORE-BUSCA(e, f)

```
1  para  $i \leftarrow 1$  até  $n + 1$  faça
2     $a[i, i - 1] \leftarrow 0$ 
3  para  $t \leftarrow 0$  até  $n - 1$  faça
4    para  $i \leftarrow 1$  até  $n - t$  faça
5       $k \leftarrow i + t$ 
6       $a[i, k] \leftarrow \sum_{j=i}^k f_j + \min_{i \leq j \leq k} \{a[i, j - 1] + a[j + 1, k]\}$ 
7  devolva  $a[1, n]$ 
```

Observações

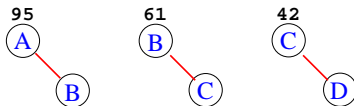
- ▶ a complexidade de tempo é $O(n^3)$
- ▶ exercício: modifique para encontrar uma solução ótima

Exemplo de solução

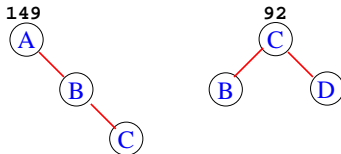
Árvores ótimas de tamanho 1



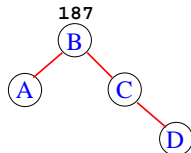
Árvores ótimas de tamanho 2



Árvores ótimas de tamanho 3



Árvores ótimas de tamanho 4



Programação dinâmica

- ▶ Problema da subsequência comum máxima

Subseqüência

Vamos trabalhar com seqüências de símbolos

- ▶ considere uma seqüência $X = \langle x_1, x_2, \dots, x_m \rangle$
- ▶ e uma outra seqüência $Z = \langle z_1, z_2, \dots, z_k \rangle$

Dizemos que Z é uma **subseqüência** de X se

- ▶ existe uma seqüência crescente de índices $\langle i_1, i_2, \dots, i_k \rangle$
- ▶ $x_{i_j} = z_j$ para cada $j = 1, 2, \dots, k$

Exemplo:

- ▶ seqüência $X = ABCDEFG$
- ▶ subseqüência $Z = ADFG$

Subseqüência

Vamos trabalhar com seqüências de símbolos

- ▶ considere uma seqüência $X = \langle x_1, x_2, \dots, x_m \rangle$
- ▶ e uma outra seqüência $Z = \langle z_1, z_2, \dots, z_k \rangle$

Dizemos que Z é uma **subseqüência** de X se

- ▶ existe uma seqüência crescente de índices $\langle i_1, i_2, \dots, i_k \rangle$
- ▶ $x_{i_j} = z_j$ para cada $j = 1, 2, \dots, k$

Exemplo:

- ▶ seqüência $X = ABCDEFG$
- ▶ subseqüência $Z = ADFG$

Subseqüência

Vamos trabalhar com seqüências de símbolos

- ▶ considere uma seqüência $X = \langle x_1, x_2, \dots, x_m \rangle$
- ▶ e uma outra seqüência $Z = \langle z_1, z_2, \dots, z_k \rangle$

Dizemos que Z é uma **subseqüência** de X se

- ▶ existe uma seqüência crescente de índices $\langle i_1, i_2, \dots, i_k \rangle$
- ▶ $x_{i_j} = z_j$ para cada $j = 1, 2, \dots, k$

Exemplo:

- ▶ seqüência $X = ABCDEFG$
- ▶ subseqüência $Z = ADFG$

Subseqüência

Vamos trabalhar com seqüências de símbolos

- ▶ considere uma seqüência $X = \langle x_1, x_2, \dots, x_m \rangle$
- ▶ e uma outra seqüência $Z = \langle z_1, z_2, \dots, z_k \rangle$

Dizemos que Z é uma **subseqüência** de X se

- ▶ existe uma seqüência crescente de índices $\langle i_1, i_2, \dots, i_k \rangle$
- ▶ $x_{i_j} = z_j$ para cada $j = 1, 2, \dots, k$

Exemplo:

- ▶ seqüência $X = ABCDEFG$
- ▶ subseqüência $Z = ADFG$

Subsequência

Vamos trabalhar com sequências de símbolos

- ▶ considere uma sequência $X = \langle x_1, x_2, \dots, x_m \rangle$
- ▶ e uma outra sequência $Z = \langle z_1, z_2, \dots, z_k \rangle$

Dizemos que Z é uma **subsequência** de X se

- ▶ existe uma sequência crescente de índices $\langle i_1, i_2, \dots, i_k \rangle$
- ▶ $x_{i_j} = z_j$ para cada $j = 1, 2, \dots, k$

Exemplo:

- ▶ sequência $X = ABCDEFG$
- ▶ subsequência $Z = ADFG$

Subseqüência

Vamos trabalhar com seqüências de símbolos

- ▶ considere uma seqüência $X = \langle x_1, x_2, \dots, x_m \rangle$
- ▶ e uma outra seqüência $Z = \langle z_1, z_2, \dots, z_k \rangle$

Dizemos que Z é uma **subseqüência** de X se

- ▶ existe uma seqüência crescente de índices $\langle i_1, i_2, \dots, i_k \rangle$
- ▶ $x_{i_j} = z_j$ para cada $j = 1, 2, \dots, k$

Exemplo:

- ▶ seqüência $X = ABCDEFG$
- ▶ subseqüência $Z = ADFG$

Subseqüência

Vamos trabalhar com seqüências de símbolos

- ▶ considere uma seqüência $X = \langle x_1, x_2, \dots, x_m \rangle$
- ▶ e uma outra seqüência $Z = \langle z_1, z_2, \dots, z_k \rangle$

Dizemos que Z é uma **subseqüência** de X se

- ▶ existe uma seqüência crescente de índices $\langle i_1, i_2, \dots, i_k \rangle$
- ▶ $x_{i_j} = z_j$ para cada $j = 1, 2, \dots, k$

Exemplo:

- ▶ seqüência $X = ABCDEFG$
- ▶ subseqüência $Z = ADFG$

Subsequência

Vamos trabalhar com sequências de símbolos

- ▶ considere uma sequência $X = \langle x_1, x_2, \dots, x_m \rangle$
- ▶ e uma outra sequência $Z = \langle z_1, z_2, \dots, z_k \rangle$

Dizemos que Z é uma **subsequência** de X se

- ▶ existe uma sequência crescente de índices $\langle i_1, i_2, \dots, i_k \rangle$
- ▶ $x_{i_j} = z_j$ para cada $j = 1, 2, \dots, k$

Exemplo:

- ▶ sequência $X = ABCDEFG$
- ▶ subsequência $Z = ADFG$

Subsequência

Vamos trabalhar com sequências de símbolos

- ▶ considere uma sequência $X = \langle x_1, x_2, \dots, x_m \rangle$
- ▶ e uma outra sequência $Z = \langle z_1, z_2, \dots, z_k \rangle$

Dizemos que Z é uma **subsequência** de X se

- ▶ existe uma sequência crescente de índices $\langle i_1, i_2, \dots, i_k \rangle$
- ▶ $x_{i_j} = z_j$ para cada $j = 1, 2, \dots, k$

Exemplo:

- ▶ sequência $X = ABCDEFG$
- ▶ subsequência $Z = ADFG$

Subsequência comum máxima

Problema:

- ▶ Entrada:
 - ▶ sequência X
 - ▶ sequência Y
- ▶ Solução:
 - ▶ subsequência comum Z de X e Y
- ▶ Objetivo:
 - ▶ maximizar o comprimento de Z

Subsequência comum máxima

Problema:

- ▶ Entrada:

- ▶ sequência X
- ▶ sequência Y

- ▶ Solução:

- ▶ subsequência comum Z de X e Y

- ▶ Objetivo:

- ▶ maximizar o comprimento de Z

Subsequência comum máxima

Problema:

- ▶ Entrada:

- ▶ sequência X
- ▶ sequência Y

- ▶ Solução:

- ▶ subsequência comum Z de X e Y

- ▶ Objetivo:

- ▶ maximizar o comprimento de Z

Subsequência comum máxima

Problema:

- ▶ Entrada:
 - ▶ sequência X
 - ▶ sequência Y
- ▶ Solução:
 - ▶ subsequência comum Z de X e Y
- ▶ Objetivo:
 - ▶ maximizar o comprimento de Z

Subsequência comum máxima

Problema:

- ▶ Entrada:

- ▶ sequência X
- ▶ sequência Y

- ▶ Solução:

- ▶ subsequência **comum** Z de X e Y

- ▶ Objetivo:

- ▶ maximizar o comprimento de Z

Subsequência comum máxima

Problema:

- ▶ Entrada:

- ▶ sequência X
- ▶ sequência Y

- ▶ Solução:

- ▶ subsequência **comum** Z de X e Y

- ▶ Objetivo:

- ▶ maximizar o comprimento de Z

Subsequência comum máxima

Problema:

- ▶ Entrada:
 - ▶ sequência X
 - ▶ sequência Y
- ▶ Solução:
 - ▶ subsequência **comum** Z de X e Y
- ▶ Objetivo:
 - ▶ **maximizar** o comprimento de Z

Subsequência comum máxima

Problema:

- ▶ Entrada:
 - ▶ sequência X
 - ▶ sequência Y
- ▶ Solução:
 - ▶ subsequência **comum** Z de X e Y
- ▶ Objetivo:
 - ▶ **maximizar** o comprimento de Z

Subestrutura ótima

Seja S uma sequência de comprimento n

- ▶ denotamos por S_i o **prefixo** de S de comprimento i
- ▶ por exemplo, se $S = ABCDEFG$, então $S_2 = AB$

Vamos estudar uma solução ótima

- ▶ considere sequências $X = \langle x_1 \dots x_m \rangle$ e $Y = \langle y_1 \dots y_n \rangle$
- ▶ a subsequência comum máxima é $Z = \langle z_1 \dots z_k \rangle$

Olhamos para os últimos elementos de X e Y

1. se $x_m = y_n$

- ▶ $z_k = x_m = y_n$ é o último elemento da solução ótima
- ▶ Z_{k-1} é subsequência comum máxima de X_{m-1} e Y_{n-1}

2. se $x_m \neq y_n$

- ▶ pelo menos x_m ou y_n não é parte da solução ótima
- ▶ ou Z é subsequência comum máxima de X_{m-1} e Y
- ▶ ou Z é subsequência comum máxima de X e Y_{n-1}

Subestrutura ótima

Seja S uma sequência de comprimento n

- ▶ denotamos por S_i o **prefixo** de S de comprimento i
- ▶ por exemplo, se $S = ABCDEFG$, então $S_2 = AB$

Vamos estudar uma solução ótima

- ▶ considere sequências $X = \langle x_1 \dots x_m \rangle$ e $Y = \langle y_1 \dots y_n \rangle$
- ▶ a subsequência comum máxima é $Z = \langle z_1 \dots z_k \rangle$

Olhamos para os últimos elementos de X e Y

1. se $x_m = y_n$
 - ▶ $z_k = x_m = y_n$ é o último elemento da solução ótima
 - ▶ Z_{k-1} é subsequência comum máxima de X_{m-1} e Y_{n-1}
2. se $x_m \neq y_n$
 - ▶ pelo menos x_m ou y_n não é parte da solução ótima
 - ▶ ou Z é subsequência comum máxima de X_{m-1} e Y
 - ▶ ou Z é subsequência comum máxima de X e Y_{n-1}

Subestrutura ótima

Seja S uma sequência de comprimento n

- ▶ denotamos por S_i o **prefixo** de S de comprimento i
- ▶ por exemplo, se $S = ABCDEFG$, então $S_2 = AB$

Vamos estudar uma solução ótima

- ▶ considere sequências $X = \langle x_1 \dots x_m \rangle$ e $Y = \langle y_1 \dots y_n \rangle$
- ▶ a subsequência comum máxima é $Z = \langle z_1 \dots z_k \rangle$

Olhamos para os últimos elementos de X e Y

1. se $x_m = y_n$

- ▶ $z_k = x_m = y_n$ é o último elemento da solução ótima
- ▶ Z_{k-1} é subsequência comum máxima de X_{m-1} e Y_{n-1}

2. se $x_m \neq y_n$

- ▶ pelo menos x_m ou y_n não é parte da solução ótima
- ▶ ou Z é subsequência comum máxima de X_{m-1} e Y
- ▶ ou Z é subsequência comum máxima de X e Y_{n-1}

Subestrutura ótima

Seja S uma sequência de comprimento n

- ▶ denotamos por S_i o **prefixo** de S de comprimento i
- ▶ por exemplo, se $S = ABCDEFG$, então $S_2 = AB$

Vamos estudar uma solução ótima

- ▶ considere sequências $X = \langle x_1 \dots x_m \rangle$ e $Y = \langle y_1 \dots y_n \rangle$
- ▶ a subsequência comum máxima é $Z = \langle z_1 \dots z_k \rangle$

Olhamos para os últimos elementos de X e Y

1. se $x_m = y_n$

- ▶ $z_k = x_m = y_n$ é o último elemento da solução ótima
- ▶ Z_{k-1} é subsequência comum máxima de X_{m-1} e Y_{n-1}

2. se $x_m \neq y_n$

- ▶ pelo menos x_m ou y_n não é parte da solução ótima
- ▶ ou Z é subsequência comum máxima de X_{m-1} e Y
- ▶ ou Z é subsequência comum máxima de X e Y_{n-1}

Subestrutura ótima

Seja S uma sequência de comprimento n

- ▶ denotamos por S_i o **prefixo** de S de comprimento i
- ▶ por exemplo, se $S = ABCDEFG$, então $S_2 = AB$

Vamos estudar uma solução ótima

- ▶ considere sequências $X = \langle x_1 \dots x_m \rangle$ e $Y = \langle y_1 \dots y_n \rangle$
- ▶ a subsequência comum máxima é $Z = \langle z_1 \dots z_k \rangle$

Olhamos para os últimos elementos de X e Y

1. se $x_m = y_n$
 - ▶ $z_k = x_m = y_n$ é o último elemento da solução ótima
 - ▶ Z_{k-1} é subsequência comum máxima de X_{m-1} e Y_{n-1}
2. se $x_m \neq y_n$
 - ▶ pelo menos x_m ou y_n não é parte da solução ótima
 - ▶ ou Z é subsequência comum máxima de X_{m-1} e Y
 - ▶ ou Z é subsequência comum máxima de X e Y_{n-1}

Subestrutura ótima

Seja S uma sequência de comprimento n

- ▶ denotamos por S_i o **prefixo** de S de comprimento i
- ▶ por exemplo, se $S = ABCDEFG$, então $S_2 = AB$

Vamos estudar uma solução ótima

- ▶ considere sequências $X = \langle x_1 \dots x_m \rangle$ e $Y = \langle y_1 \dots y_n \rangle$
- ▶ a subsequência comum máxima é $Z = \langle z_1 \dots z_k \rangle$

Olhamos para os últimos elementos de X e Y

1. se $x_m = y_n$

- ▶ $z_k = x_m = y_n$ é o último elemento da solução ótima
- ▶ Z_{k-1} é subsequência comum máxima de X_{m-1} e Y_{n-1}

2. se $x_m \neq y_n$

- ▶ pelo menos x_m ou y_n não é parte da solução ótima
- ▶ ou Z é subsequência comum máxima de X_{m-1} e Y
- ▶ ou Z é subsequência comum máxima de X e Y_{n-1}

Subestrutura ótima

Seja S uma sequência de comprimento n

- ▶ denotamos por S_i o **prefixo** de S de comprimento i
- ▶ por exemplo, se $S = ABCDEFG$, então $S_2 = AB$

Vamos estudar uma solução ótima

- ▶ considere sequências $X = \langle x_1 \dots x_m \rangle$ e $Y = \langle y_1 \dots y_n \rangle$
- ▶ a subsequência comum máxima é $Z = \langle z_1 \dots z_k \rangle$

Olhamos para os últimos elementos de X e Y

1. se $x_m = y_n$
 - ▶ $z_k = x_m = y_n$ é o último elemento da solução ótima
 - ▶ Z_{k-1} é subsequência comum máxima de X_{m-1} e Y_{n-1}
2. se $x_m \neq y_n$
 - ▶ pelo menos x_m ou y_n não é parte da solução ótima
 - ▶ ou Z é subsequência comum máxima de X_{m-1} e Y
 - ▶ ou Z é subsequência comum máxima de X e Y_{n-1}

Subestrutura ótima

Seja S uma sequência de comprimento n

- ▶ denotamos por S_i o **prefixo** de S de comprimento i
- ▶ por exemplo, se $S = ABCDEFG$, então $S_2 = AB$

Vamos estudar uma solução ótima

- ▶ considere sequências $X = \langle x_1 \dots x_m \rangle$ e $Y = \langle y_1 \dots y_n \rangle$
- ▶ a subsequência comum máxima é $Z = \langle z_1 \dots z_k \rangle$

Olhamos para os últimos elementos de X e Y

1. se $x_m = y_n$

- ▶ $z_k = x_m = y_n$ é o último elemento da solução ótima
- ▶ Z_{k-1} é subsequência comum máxima de X_{m-1} e Y_{n-1}

2. se $x_m \neq y_n$

- ▶ pelo menos x_m ou y_n não é parte da solução ótima
- ▶ ou Z é subsequência comum máxima de X_{m-1} e Y
- ▶ ou Z é subsequência comum máxima de X e Y_{n-1}

Subestrutura ótima

Seja S uma sequência de comprimento n

- ▶ denotamos por S_i o **prefixo** de S de comprimento i
- ▶ por exemplo, se $S = ABCDEFG$, então $S_2 = AB$

Vamos estudar uma solução ótima

- ▶ considere sequências $X = \langle x_1 \dots x_m \rangle$ e $Y = \langle y_1 \dots y_n \rangle$
- ▶ a subsequência comum máxima é $Z = \langle z_1 \dots z_k \rangle$

Olhamos para os últimos elementos de X e Y

1. se $x_m = y_n$

- ▶ $z_k = x_m = y_n$ é o último elemento da solução ótima
- ▶ Z_{k-1} é subsequência comum máxima de X_{m-1} e Y_{n-1}

2. se $x_m \neq y_n$

- ▶ pelo menos x_m ou y_n não é parte da solução ótima
- ▶ ou Z é subsequência comum máxima de X_{m-1} e Y
- ▶ ou Z é subsequência comum máxima de X e Y_{n-1}

Subestrutura ótima

Seja S uma sequência de comprimento n

- ▶ denotamos por S_i o **prefixo** de S de comprimento i
- ▶ por exemplo, se $S = ABCDEFG$, então $S_2 = AB$

Vamos estudar uma solução ótima

- ▶ considere sequências $X = \langle x_1 \dots x_m \rangle$ e $Y = \langle y_1 \dots y_n \rangle$
- ▶ a subsequência comum máxima é $Z = \langle z_1 \dots z_k \rangle$

Olhamos para os últimos elementos de X e Y

1. se $x_m = y_n$
 - ▶ $z_k = x_m = y_n$ é o último elemento da solução ótima
 - ▶ Z_{k-1} é subsequência comum máxima de X_{m-1} e Y_{n-1}
2. se $x_m \neq y_n$
 - ▶ pelo menos x_m ou y_n não é parte da solução ótima
 - ▶ ou Z é subsequência comum máxima de X_{m-1} e Y
 - ▶ ou Z é subsequência comum máxima de X e Y_{n-1}

Subestrutura ótima

Seja S uma sequência de comprimento n

- ▶ denotamos por S_i o **prefixo** de S de comprimento i
- ▶ por exemplo, se $S = ABCDEFG$, então $S_2 = AB$

Vamos estudar uma solução ótima

- ▶ considere sequências $X = \langle x_1 \dots x_m \rangle$ e $Y = \langle y_1 \dots y_n \rangle$
- ▶ a subsequência comum máxima é $Z = \langle z_1 \dots z_k \rangle$

Olhamos para os últimos elementos de X e Y

1. se $x_m = y_n$
 - ▶ $z_k = x_m = y_n$ é o último elemento da solução ótima
 - ▶ Z_{k-1} é subsequência comum máxima de X_{m-1} e Y_{n-1}
2. se $x_m \neq y_n$
 - ▶ pelo menos x_m ou y_n não é parte da solução ótima
 - ▶ ou Z é subsequência comum máxima de X_{m-1} e Y
 - ▶ ou Z é subsequência comum máxima de X e Y_{n-1}

Subestrutura ótima

Seja S uma sequência de comprimento n

- ▶ denotamos por S_i o **prefixo** de S de comprimento i
- ▶ por exemplo, se $S = ABCDEFG$, então $S_2 = AB$

Vamos estudar uma solução ótima

- ▶ considere sequências $X = \langle x_1 \dots x_m \rangle$ e $Y = \langle y_1 \dots y_n \rangle$
- ▶ a subsequência comum máxima é $Z = \langle z_1 \dots z_k \rangle$

Olhamos para os últimos elementos de X e Y

1. se $x_m = y_n$
 - ▶ $z_k = x_m = y_n$ é o último elemento da solução ótima
 - ▶ Z_{k-1} é subsequência comum máxima de X_{m-1} e Y_{n-1}
2. se $x_m \neq y_n$
 - ▶ pelo menos x_m ou y_n não é parte da solução ótima
 - ▶ ou Z é subsequência comum máxima de X_{m-1} e Y
 - ▶ ou Z é subsequência comum máxima de X e Y_{n-1}

Subestrutura ótima

Seja S uma sequência de comprimento n

- ▶ denotamos por S_i o **prefixo** de S de comprimento i
- ▶ por exemplo, se $S = ABCDEFG$, então $S_2 = AB$

Vamos estudar uma solução ótima

- ▶ considere sequências $X = \langle x_1 \dots x_m \rangle$ e $Y = \langle y_1 \dots y_n \rangle$
- ▶ a subsequência comum máxima é $Z = \langle z_1 \dots z_k \rangle$

Olhamos para os últimos elementos de X e Y

1. se $x_m = y_n$
 - ▶ $z_k = x_m = y_n$ é o último elemento da solução ótima
 - ▶ Z_{k-1} é subsequência comum máxima de X_{m-1} e Y_{n-1}
2. se $x_m \neq y_n$
 - ▶ pelo menos x_m ou y_n não é parte da solução ótima
 - ▶ ou Z é subsequência comum máxima de X_{m-1} e Y
 - ▶ ou Z é subsequência comum máxima de X e Y_{n-1}

Subestrutura ótima

Seja S uma sequência de comprimento n

- ▶ denotamos por S_i o **prefixo** de S de comprimento i
- ▶ por exemplo, se $S = ABCDEFG$, então $S_2 = AB$

Vamos estudar uma solução ótima

- ▶ considere sequências $X = \langle x_1 \dots x_m \rangle$ e $Y = \langle y_1 \dots y_n \rangle$
- ▶ a subsequência comum máxima é $Z = \langle z_1 \dots z_k \rangle$

Olhamos para os últimos elementos de X e Y

1. se $x_m = y_n$
 - ▶ $z_k = x_m = y_n$ é o último elemento da solução ótima
 - ▶ Z_{k-1} é subsequência comum máxima de X_{m-1} e Y_{n-1}
2. se $x_m \neq y_n$
 - ▶ pelo menos x_m ou y_n não é parte da solução ótima
 - ▶ ou Z é subsequência comum máxima de X_{m-1} e Y
 - ▶ ou Z é subsequência comum máxima de X e Y_{n-1}

Subproblema

Definimos o seguinte **subproblema**

- ▶ considere $i = 0, 1, \dots, m$ e $j = 0, 1, \dots, n$
- ▶ seja $c[i, j]$ o comprimento da subsequência comum mais longa dos prefixos X_i e Y_j

Podemos utilizar a seguinte recorrência

$$c[i, j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0 \\ c[i-1, j-1] + 1 & \text{se } x_i = y_j \\ \max\{c[i-1, j], c[i, j-1]\} & \text{se } x_i \neq y_j \end{cases}$$

Subproblema

Definimos o seguinte **subproblema**

- ▶ considere $i = 0, 1, \dots, m$ e $j = 0, 1, \dots, n$
- ▶ seja $c[i, j]$ o comprimento da subsequência comum mais longa dos prefixos X_i e Y_j

Podemos utilizar a seguinte recorrência

$$c[i, j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0 \\ c[i-1, j-1] + 1 & \text{se } x_i = y_j \\ \max\{c[i-1, j], c[i, j-1]\} & \text{se } x_i \neq y_j \end{cases}$$

Subproblema

Definimos o seguinte **subproblema**

- ▶ considere $i = 0, 1, \dots, m$ e $j = 0, 1, \dots, n$
- ▶ seja $c[i, j]$ o comprimento da subsequência comum mais longa dos prefixos X_i e Y_j

Podemos utilizar a seguinte recorrência

$$c[i, j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0 \\ c[i-1, j-1] + 1 & \text{se } x_i = y_j \\ \max\{c[i-1, j], c[i, j-1]\} & \text{se } x_i \neq y_j \end{cases}$$

Subproblema

Definimos o seguinte **subproblema**

- ▶ considere $i = 0, 1, \dots, m$ e $j = 0, 1, \dots, n$
- ▶ seja $c[i, j]$ o comprimento da subsequência comum mais longa dos prefixos X_i e Y_j

Podemos utilizar a seguinte recorrência

$$c[i, j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0 \\ c[i-1, j-1] + 1 & \text{se } x_i = y_j \\ \max\{c[i-1, j], c[i, j-1]\} & \text{se } x_i \neq y_j \end{cases}$$

Subproblema

Definimos o seguinte **subproblema**

- ▶ considere $i = 0, 1, \dots, m$ e $j = 0, 1, \dots, n$
- ▶ seja $c[i, j]$ o comprimento da subsequência comum mais longa dos prefixos X_i e Y_j

Podemos utilizar a seguinte recorrência

$$c[i, j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0 \\ c[i - 1, j - 1] + 1 & \text{se } x_i = y_j \\ \max\{c[i - 1, j], c[i, j - 1]\} & \text{se } x_i \neq y_j \end{cases}$$

Algoritmo de programação dinâmica

SCM(X, m, Y, n)

```
1  para  $i \leftarrow 0$  até  $m$  faça
2     $c[i, 0] \leftarrow 0$ 
3  para  $j \leftarrow 1$  até  $n$  faça
4     $c[0, j] \leftarrow 0$ 
5  para  $i \leftarrow 1$  até  $m$  faça
6    para  $j = 1$  até  $n$  faça
7      se  $x_i = y_j$  então
8         $c[i, j] \leftarrow c[i - 1, j - 1] + 1$ 
9         $b[i, j] \leftarrow "\diagdown"$ 
10     senão se  $c[i, j - 1] > c[i - 1, j]$  então
11        $c[i, j] \leftarrow c[i, j - 1]$ 
12        $b[i, j] \leftarrow "\leftarrow"$ 
13     senão
14        $c[i, j] \leftarrow c[i - 1, j]$ 
15        $b[i, j] \leftarrow "\uparrow"$ 
16  devolva  $c[m, n]$ 
```

- ▶ a tabela b guarda quais subproblemas foram escolhidos

Algoritmo de programação dinâmica

```
SCM( $X, m, Y, n$ )
1  para  $i \leftarrow 0$  até  $m$  faça
2     $c[i, 0] \leftarrow 0$ 
3  para  $j \leftarrow 1$  até  $n$  faça
4     $c[0, j] \leftarrow 0$ 
5  para  $i \leftarrow 1$  até  $m$  faça
6    para  $j = 1$  até  $n$  faça
7      se  $x_i = y_j$  então
8         $c[i, j] \leftarrow c[i - 1, j - 1] + 1$ 
9         $b[i, j] \leftarrow "\diagdown"$ 
10     senão se  $c[i, j - 1] > c[i - 1, j]$  então
11        $c[i, j] \leftarrow c[i, j - 1]$ 
12        $b[i, j] \leftarrow "\leftarrow"$ 
13     senão
14        $c[i, j] \leftarrow c[i - 1, j]$ 
15        $b[i, j] \leftarrow "\uparrow"$ 
16  devolva  $c[m, n]$ 
```

- ▶ a tabela b guarda quais subproblemas foram escolhidos

Exemplo de tabelas preenchidas

Exemplo para $X = abcb$ e $Y = bdcab$.

	Y	b	d	c	a	b
X	0	1	2	3	4	5
0	0	0	0	0	0	0
a	1	0	0	0	1	1
b	2	0	1	1	1	2
c	3	0	1	1	2	2
b	4	0	1	1	2	3

	Y	(b)	d	(c)	a	(b)	
X	0	1	2	3	4	5	
0							
a	1		↑	↑	↑	↖	←
(b)	2		↖	←	←	↑	↖
(c)	3		↑	↑	↖	←	↑
(b)	4		↖	↑	↑	↑	↖

Complexidade de tempo

- ▶ cada entrada da tabela é preenchido em tempo contante
- ▶ assim, o algoritmo gasta tempo $O(mn)$

Complexidade de espaço

- ▶ o algoritmo gasta tabelas de tamanho total $O(mn)$
- ▶ podemos manter apenas duas linhas ou colunas
- ▶ o algoritmo melhorado usa memória $O(\min\{m, n\})$
- ▶ mas manter a tabela b permite encontrar a solução

Complexidade de tempo

- ▶ cada entrada da tabela é preenchido em tempo contante
- ▶ assim, o algoritmo gasta tempo $O(mn)$

Complexidade de espaço

- ▶ o algoritmo gasta tabelas de tamanho total $O(mn)$
- ▶ podemos manter apenas duas linhas ou colunas
- ▶ o algoritmo melhorado usa memória $O(\min\{m, n\})$
- ▶ mas manter a tabela b permite encontrar a solução

Complexidade de tempo

- ▶ cada entrada da tabela é preenchido em tempo contante
- ▶ assim, o algoritmo gasta tempo $O(mn)$

Complexidade de espaço

- ▶ o algoritmo gasta tabelas de tamanho total $O(mn)$
- ▶ podemos manter apenas duas linhas ou colunas
- ▶ o algoritmo melhorado usa memória $O(\min\{m, n\})$
- ▶ mas manter a tabela b permite encontrar a solução

Complexidade de tempo

- ▶ cada entrada da tabela é preenchido em tempo contante
- ▶ assim, o algoritmo gasta tempo $O(mn)$

Complexidade de espaço

- ▶ o algoritmo gasta tabelas de tamanho total $O(mn)$
- ▶ podemos manter apenas duas linhas ou colunas
- ▶ o algoritmo melhorado usa memória $O(\min\{m, n\})$
- ▶ mas manter a tabela b permite encontrar a solução

Complexidade de tempo

- ▶ cada entrada da tabela é preenchido em tempo contante
- ▶ assim, o algoritmo gasta tempo $O(mn)$

Complexidade de espaço

- ▶ o algoritmo gasta tabelas de tamanho total $O(mn)$
- ▶ podemos manter apenas duas linhas ou colunas
- ▶ o algoritmo melhorado usa memória $O(\min\{m, n\})$
- ▶ mas manter a tabela b permite encontrar a solução

Complexidade de tempo

- ▶ cada entrada da tabela é preenchido em tempo contante
- ▶ assim, o algoritmo gasta tempo $O(mn)$

Complexidade de espaço

- ▶ o algoritmo gasta tabelas de tamanho total $O(mn)$
- ▶ podemos manter apenas duas linhas ou colunas
- ▶ o algoritmo melhorado usa memória $O(\min\{m, n\})$
- ▶ mas manter a tabela b permite encontrar a solução

Complexidade de tempo

- ▶ cada entrada da tabela é preenchido em tempo contante
- ▶ assim, o algoritmo gasta tempo $O(mn)$

Complexidade de espaço

- ▶ o algoritmo gasta tabelas de tamanho total $O(mn)$
- ▶ podemos manter apenas duas linhas ou colunas
- ▶ o algoritmo melhorado usa memória $O(\min\{m, n\})$
- ▶ mas manter a tabela b permite encontrar a solução

Complexidade de tempo

- ▶ cada entrada da tabela é preenchido em tempo contante
- ▶ assim, o algoritmo gasta tempo $O(mn)$

Complexidade de espaço

- ▶ o algoritmo gasta tabelas de tamanho total $O(mn)$
- ▶ podemos manter apenas duas linhas ou colunas
- ▶ o algoritmo melhorado usa memória $O(\min\{m, n\})$
- ▶ mas manter a tabela b permite encontrar a solução

Recuperando uma solução

```
RECUPERA-SCM( $b, X, i, j$ )  
1  se  $i = 0$  ou  $j = 0$  então  
2    retorne  
3  se  $b[i, j] = \nwarrow$  então  
4    RECUPERA-SCM( $b, X, i - 1, j - 1$ )  
5    imprima  $x_i$   
6  senão se  $b[i, j] = \uparrow$  então  
7    RECUPERA-SCM( $b, X, i - 1, j$ )  
8  senão  
9    RECUPERA-SCM( $b, X, i, j - 1$ )
```

- ▶ a chamada inicial é **RECUPERA-SCM**(b, X, m, n)

Recuperando uma solução

```
RECUPERA-SCM( $b, X, i, j$ )  
1  se  $i = 0$  ou  $j = 0$  então  
2    retorne  
3  se  $b[i, j] = "\nwarrow"$  então  
4    RECUPERA-SCM( $b, X, i - 1, j - 1$ )  
5    imprima  $x_i$   
6  senão se  $b[i, j] = "\uparrow"$  então  
7    RECUPERA-SCM( $b, X, i - 1, j$ )  
8  senão  
9    RECUPERA-SCM( $b, X, i, j - 1$ )
```

- ▶ a chamada inicial é **RECUPERA-SCM**(b, X, m, n)