Projeto e Análise de Algoritmos

Programação dinâmica

Lehilton Pedrosa

Primeiro Semestre de 2020

Programação dinâmica

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

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

Programação dinâmica

- Multiplicação de cadeias de matrizes

Parentização

Considere um produto de matrizes

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

- 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:

$$
\begin{array}{lcl}
M = (M_1 \times (M_2 \times (M_3 \times M_4))) & \longrightarrow & 5.300 \text{ multiplicações} \\
M = (M_1 \times ((M_2 \times M_3) \times M_4)) & \longrightarrow & 3.400 \text{ multiplicações} \\
M = ((M_1 \times M_2) \times (M_3 \times M_4)) & \longrightarrow & 4.500 \text{ multiplicações} \\
M = ((M_1 \times (M_2 \times M_3)) \times M_4) & \longrightarrow & 29.200 \text{ multiplicações} \\
M = (((M_1 \times M_2) \times M_3) \times M_4) & \longrightarrow & 152.000 \text{ multiplicações}
\end{array}
$$

- 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

Encontrando uma subestrututra ótima

Considere uma parentização ótima

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

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

- 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 multiplações para computar $M_{i,k}$ e $M_{k+1,n}$ é mínimo

Encontramos uma substrutura 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

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

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

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

Algoritmo recursivo

Minimo-Multiplicacoes-Recursivo(b, i, j)

se $i = j$ **então**
    $m[i,j] \leftarrow 0$
senão
    $m[i,j] \leftarrow \infty$
    para $k \leftarrow i$ **até** $j - 1$ **faça**
        $q \leftarrow$ Minimo-Multiplicacoes-Recursivo(b, i, k)
            $+$ Minimo-Multiplicacoes-Recursivo$(b, k+1, j)$
            $+ b[i-1] \cdot b[k] \cdot b[j]$
        se $m > q$ **então**
            $m[i,j] \leftarrow q$
            $s[i,j] \leftarrow k$
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)
    se $i = j$ **então**
        devolva M_i
    senão
        $X \leftarrow$ MULTIPLICA-MATRIZES$(M, s, i, s[i,j])$
        $Y \leftarrow$ MULTIPLICA-MATRIZES$(M, s, s[i,j]+1, j)$
        devolva $\mathit{Multiplica}(X, Y, b[i-1], b[s[i,j]], b[j])$

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 (cont)

Analisando com calma

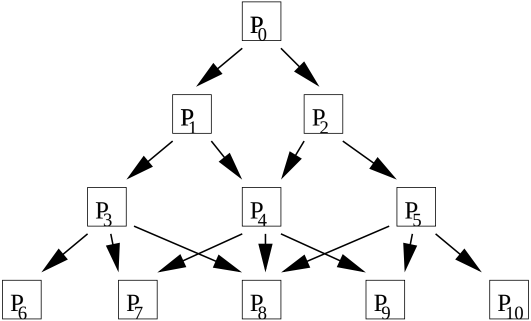
- o algoritmo recalcula o mesmo valor $m[i,j]$ várias vezes
- por exemplo, se $n = 4$, os valores $m[1,2]$, $m[2,3]$ e $m[3,4]$ são computados duas vezes

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

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)

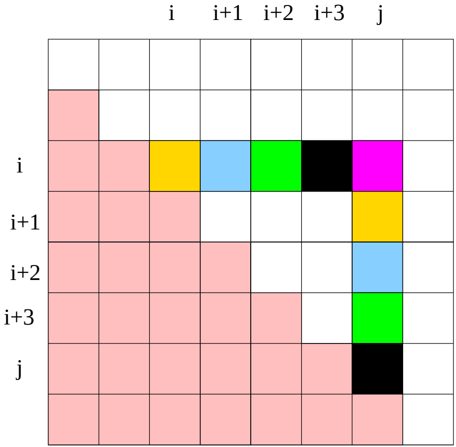
```
se m[i, j] não está definido então
    se i = j então
        m[i, j] ← 0
    senão
        m[i, j] ← ∞
        para k ← i até j − 1 faça
            q ← Minimo-Multiplicacoes-Memorizado(b, i, k)
                + Minimo-Multiplicacoes-Memorizado(b, k + 1, j)
                + b[i − 1] · b[k] · b[j]
            se m > q então
                m[i, j] ← q
                s[i, j] ← k
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)
para i ← 1 até n faça
    m[i,i] ← 0
para u ← 1 até n − 1 faça
    para i ← 1 até n − u faça
        j ← i + u
        m[i,j] ← ∞
        para k ← i até j − 1 faça
            q ← m[i,k] + m[k+1,j] + b[i−1] · b[k] · b[j]
            se q < m[i,j] então
                m[i,j] ← q
                s[i,j] ← k
devolva m[1,n]
```

Exemplo de tabela



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 }

b0*b1*b3=200*2*20=8000

b0*b2*b3=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

	1	2	3	4
1	_	1	1	1
2		_	2	3
3			_	3
4				_

s

b0*b1*b4=200*2*5=2000

b0*b2*b4=200*30*5=30000

b0*b3*b4=200*20*5=20000

{ 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 × 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 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 $I \subseteq \{1, 2, \ldots, n\}$ 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 $i = 1, \ldots, n$

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

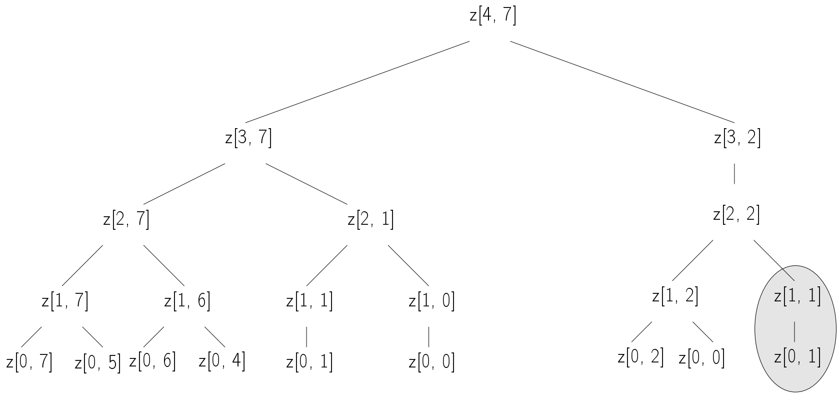
- seja d a capacidade residual da mochila
- considere os k primeiros itens
- defina $z[k,d]$ o valor máximo de um subconjunto $I' \subseteq \{1,2,\dots,k\}$ tal que $\sum_{i \in I'} w_i \leq d$

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

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

Sobreposição de subproblemas

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



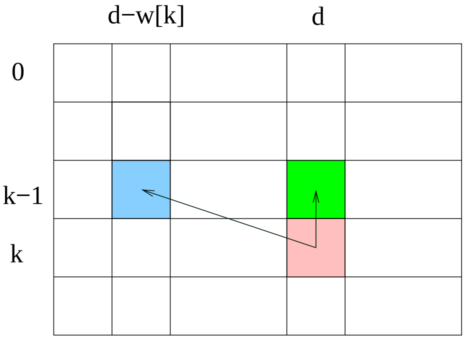
O subproblema $z[1, 1]$ é computado duas vezes.

Programação dinâmica

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 cáculo 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 $z[k-1, d-w_k]$.

Algoritmo de programação dinâmica

Mochila(W, w, c, n)

para $d \leftarrow 0$ **até** W **faça**

  $z[0, d] \leftarrow 0$

para $k \leftarrow 1$ **até** n **faça**

  para $d \leftarrow 0$ **até** W **faça**

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

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

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

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 \ 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 \ 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 \ 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 \ 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 \ 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 \ 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

Análise

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)

```
d ← W
para k ← n decrescendo até 1 faça
    se z[k, d] = z[k − 1, d] então
        x[k] ← 0
    senão
        x[k] ← 1
        d ← d − w_k
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 \ 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 \ 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 \ 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 \ 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 \ 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 , x[2] = x[3] = 0

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 $e_1 < e_2 < \cdots < e_n$
- 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

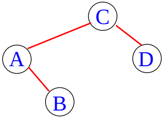
Pergunta: qual árvore minimiza o número de acessos a nós?

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 $f_D = 12$

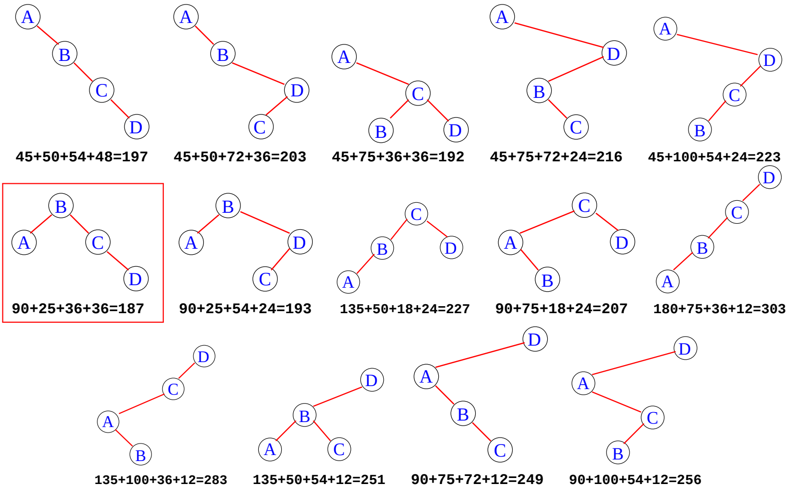
Exemplo de árvore de busca:



90+75+18+24=207

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

Listando todas as árvores de busca



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

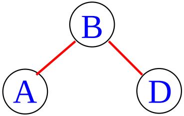
Problema da árvore de busca ótima

Problema:

- Entrada:
  - sequência de chaves $e_1 < e_2 < \cdots < e_n$
  - frequências de consulta f_i para cada chave e_i
- Solução:
  - árvore binária de busca com nós $e_1, e_2, \ldots, e_n$
- 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

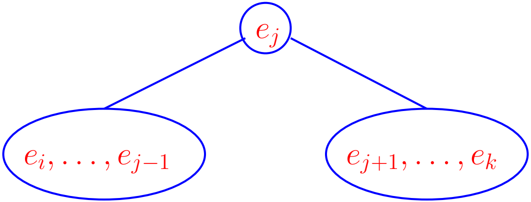
Denote por $T(v)$ a subárvore enraizada em v

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

Subestrutura ótima

Considere uma árvore de busca T com chave e_j

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



- subárvores de $e_i, \ldots, e_{j-1}$ e $e_{j+1}, \ldots, e_k$ devem ser ótimas

Decompondo em subproblema

- qualquer um de $e_i, \ldots, e_k$ 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 $e_i, \ldots, e_k$

Podemos utilizar a recorrência:

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

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)

para $i \leftarrow 1$ **até** $n+1$ **faça**
    $a[i, i-1] \leftarrow 0$
para $t \leftarrow 0$ **até** $n-1$ **faça**
    para $i \leftarrow 1$ **até** $n-t$ **faça**
        $k \leftarrow i + t$
        $a[i,k] \leftarrow \sum_{j=i}^{k} f_j + \min_{i \le j \le k} \left\{ a[i, j-1] + a[j+1, k] \right\}$
devolva $a[1, n]$

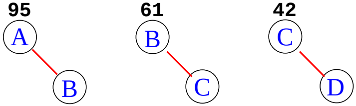
Observações

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

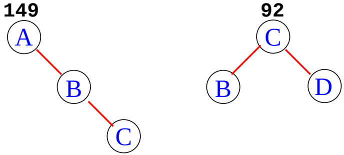
Exemplo de solução

Árvores ótimas de tamanho 1

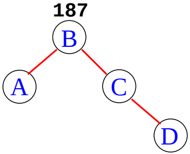
Árvores ótimas de tamanho 2



Árvores ótimas de tamanho 3



Árvores ótimas de tamanho 4



Programação dinâmica

- Problema da subsequência comum máxima

Subsequência

Vamos trabalhar com sequências de símbolos

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

Dizemos que Z é uma subsequência de X se

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

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

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

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

Podemos utilizar a seguinte recorrência

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

Algoritmo de programação dinâmica

```
SCM(X, m, Y, n)
    para i ← 0 até m faça
        c[i, 0] ← 0
    para j ← 1 até n faça
        c[0, j] ← 0
    para i ← 1 até m faça
        para j = 1 até n faça
            se x_i = y_j então
                c[i, j] ← c[i − 1, j − 1] + 1
                b[i, j] ← "↖"
            senão se c[i, j − 1] > c[i − 1, j] então
                c[i, j] ← c[i, j − 1]
                b[i, j] ← "←"
            senão
                c[i, j] ← c[i − 1, j]
                b[i, j] ← "↑"
    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	0	1	1
b	2	0	1	1	1	1	2
c	3	0	1	1	2	2	2
b	4	0	1	1	2	2	3

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

Análise

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)

- **se** $i = 0$ ou $j = 0$ **então**
  - **retorne**
- **se** $b[i,j] =$ "$\nwarrow$" **então**
  - Recupera-SCM$(b, X, i-1, j-1)$
  - **imprima** x_i
- **senão se** $b[i,j] =$ "$\uparrow$" **então**
  - Recupera-SCM$(b, X, i-1, j)$
- **senão**
  - Recupera-SCM$(b, X, i, j-1)$

- a chamada inicial é Recupera-SCM(b, X, m, n)