Programação Dinâmica

André Gustavo dos Santos
DPI - UFV

Características

- Técnica de programação que pode acelerar bastante a resolução de alguns problemas.
- Há problemas que podem ser resolvidos com algoritmos simples, diretos, porém exponencias
- Há outros com algoritmos recursivos ou backtracking, às vezes difíceis e ainda exponenciais
- Para alguns desses, existe método utilizando programação dinâmica, polinomial ou pseudo-polinomial

Características

- Pré-requisitos
  - Optimal substructure: solução ótima do problema pode ser obtida a partir da solução ótima de subproblemas
  - Overlapping subproblems: vários problemas maiores precisam da solução dos mesmos subproblemas
- Porque não usar recursão simples?
  - O mesmo subproblema seria resolvido várias vezes
  - O algoritmo poderia ficar exponencial
- Idéia geral
  - *Memoization*: manter uma tabela com resultados de problemas menores
  - Utilizar os resultados para solucionar problemas maiores

Características

- Em campeonatos de programação
  - Pode ser a chave para resolver um problema
  - Pode ser o truque para resolver problemas que não passariam no tempo limite com outras técnicas
  - Geralmente um ou mais problemas precisam de PD
    - Pode significar um balão a mais :)

Características

- Solução “top down”
  - Dividir o problema em subproblemas menores
  - Guardar soluções dos subproblemas em uma tabela à medida que são resolvidos
  - Só resolver um subproblema se sua solução ainda não está guardada na tabela
  - Recursão + *memoization*
- Solução “bottom up”
  - Resolver antecipadamente os subproblemas que podem ser necessários
  - Resolver os problemas em “ordem de tamanho” guardando os resultados em uma tabela
  - Ao resolver um problema, seus subproblemas já foram resolvidos

Exemplo – Fibonacci (sem PD)

```
int fibo (int n)
{
  if ((n==0)||(n==1))
    return n;
  else
    return fibo(n-1) + fibo(n-2);
}
```

- Muitos cálculos desnecessários
  - Imagine quantas vezes fibo(1) é chamado!
- Para *n* grande pode ficar lento

Exemplo – Fibonacci (com PD top down)

```
int tab[MAX]; //contendo 0,1,-1,-1,-1,-1,-1,...

int fibo (int n)
{
  if (tab[n]==-1)
     tab[n] = fibo(n-1) + fibo(n-2);
  return tab[n];
}
```

- *fibo(n)* é calculado uma única vez para cada valor de *n*

Exemplo – Fibonacci (com PD bottom up)

```
int tab[MAX];

int fibo (int n)
{
  tab[0] = 0;
  tab[1] = 1;

  for(int i=2;i<=n;i++)
    tab[n] = tab[n-1] + tab[n-2];
  return tab[n];
}
```

- Elimina a recursividade

Comparação (de uma forma geral)

- "Botom up"
  - elimina a recursividade, mas preenche a tabela inteira
- "Top down"
  - só calcula realmente o que precisa, utilizando recursividade
- Qual delas utilizar?
  - A que você tiver mais habilidade
  - A que você conseguir codificar o problema
  - A que for mais fácil para codificar o problema
  - A que for mais rápida para o problema
  - A que funcionar primeiro...

Selos

- Dados
  - *N* valores de selos que podem ser usados
  - Valor *T* de uma carta a ser selada
- Objetivo
  - Descobrir se é possível selar a carta com o valor exato
- Exemplo
  - Selos: 5, 7, 13 e T = 19
  - Selos: 5, 7, 13 e T = 16

Selos

- Dados
  - *N* valores de selos que podem ser usados
  - Valor *T* de uma carta a ser selada
- Objetivo
  - Descobrir se é possível selar a carta com o valor exato
- Exemplo
  - Selos: 5, 7, 13 e T = 19 - SIM: 5 + 7 + 7
  - Selos: 5, 7, 13 e T = 16 - NÃO

Selos – recursivo

```
bool valor(int v[], int n, int t)
{
  if (t==0)
    return true;
  if (t<0)
    return false;
  for(int i=0; i<n; i++)
    if (valor(v,n,t-v[i])
      return true;
  return false;
}
```

Selos – programação dinâmica

```
bool valor(int v[], int n, int t)
{
 bool possivel[MAXT+1] = {false};

 possivel[0] = true;
 int i = 0;
 while(!possivel[t] && i<t) {
    if (possivel[i])              //se é possivel valor i
       for(int j=0; j<n; j++)
          if (i+v[j] <= MAXT)     //também é i + cada selo
             possivel[i+v[j]] = true;
    i++;
 }
 return possivel[t];
}
```

Selos (sem repetição)

- Dados
  - *N* selos que podem ser usados, cada um com um valor
  - Valor *T* de uma carta a ser selada
- Objetivo
  - Descobrir se é possível selar a carta com o valor exato
- Exemplo
  - Selos: 5, 5, 7, 13 e T = 25
  - Selos: 5, 5, 7, 13 e T = 26
  - Selos: 5, 5, 7, 13 e T = 23

Selos (sem repetição)

- Dados
  - *N* selos que podem ser usados, cada um com um valor
  - Valor *T* de uma carta a ser selada
- Objetivo
  - Descobrir se é possível selar a carta com o valor exato
- Exemplo
  - Selos: 5, 5, 7, 13 e T = 25 - SIM: 5 + 7 + 13
  - Selos: 5, 5, 7, 13 e T = 26 - NÃO
  - Selos: 5, 5, 7, 13 e T = 23 - SIM: 5 + 5 + 13

Selos (sem repetição) – programação dinâmica

- Ordenar os valores dos selos previamente
- Para cada valor de selo, do menor para o maior
  - Para cada valor possível (do último marcado até o zero)
  - Somar o valor do selo e marcar o valor como possível

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 .. 25 26 27 28 29 30
 X  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -     -  -  -  -  -  -
selo 5
 X  -  -  -  -  X  -  -  -  -  -  -  -  -  -  -  -  -  -     -  -  -  -  -  -
selo 5
 X  -  -  -  -  X  -  -  -  -  X  -  -  -  -  -  -  -  -     -  -  -  -  -  -
```

Selos (sem repetição) – programação dinâmica

- Ordenar os valores dos selos previamente
- Para cada valor de selo, do menor para o maior
  - Para cada valor possível (do último marcado até o zero)
  - Somar o valor do selo e marcar o valor como possível

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 .. 25 26 27 28 29 30
 X  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -     -  -  -  -  -  -
selo 5
 X  -  -  -  -  X  -  -  -  -  -  -  -  -  -  -  -  -  -     -  -  -  -  -  -
selo 5
 X  -  -  -  -  X  -  -  -  -  X  -  -  -  -  -  -  -  -     -  -  -  -  -  -
selo 7
 X  -  -  -  -  X  -  X  -  -  X  -  X  -  -  -  -  X  -     -  -  -  -  -  -
selo 13
 X  -  -  -  -  X  -  X  -  -  X  -  X  X  -  -  -  X  X     X  -  -  -  -  X
```

Selos – imprimir os valores

- Os códigos dizem se é possível atingir o valor exato
- Mas não mostram como
- Regra geral
  - Usar uma outra tabela para marcar os passos feitos

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	..	25	26	27	28	29	30
X	-	-	-	-	X	-	X	-	-	X	-	X	X	-	-	-	X	X		X	-	-	-	-	X
0	-	-	-	-	5	-	7	-	-	5	-	7	13	-	-	-	7	13	..	13	-	-	-	-	13

Selos – imprimir os valores

- Os códigos dizem se é possível atingir o valor exato
- Mas não mostram como
- Regra geral
  - Usar uma outra tabela para marcar os passos feitos

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	..	25	26	27	28	29	30
X	-	-	-	-	X	-	X	-	-	X	-	X	X	-	-	-	X	X		X	-	-	-	-	X
0	-	-	-	-	5	-	7	-	-	5	-	7	13	-	-	-	7	13	..	13	-	-	-	-	13

  - Um código de “trás pra frente” ou recursivo recupera os passos
  - Para 30: selo 13

30-13=17: selo 7

17-7=10: selo 5

10-5=5: selo 5

5-5=0: -

Counting Change

- Dados
  - Uma lista de *N* valores de moeda
  - Um valor *T* a ser trocado nessas moedas
- Objetivo
  - Calcular o número de maneiras de se fazer o troco
- Exemplo
  - Moedas de 1, 5 e 10, e troco *T* = 20

Counting Change

- Dados
  - Uma lista de *N* valores de moeda
  - Um valor *T* a ser trocado nessas moedas
- Objetivo
  - Calcular o número de maneiras de se fazer o troco
- Exemplo
  - Moedas de 1, 5 e 10, e troco *T* = 20 (9 maneiras)
    - 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
    - 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 5
    - 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 5 + 5
    - 1 + 1 + 1 + 1 + 1 + 5 + 5 + 5
    - 5 + 5 + 5 + 5
    - 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 10
    - 1 + 1 + 1 + 1 + 1 + 5 + 10
    - 5 + 5 + 10
    - 10 + 10

Counting Change – recursão

- Seja Total($V[\,],N,T$) o número de maneiras de trocar o valor T nas N moedas de valor $V_1, V_2, ..., V_N$

- Total($V[\,],N,T$) =
  - Total($V[\,],N\text{-}1,T$) + Total($V[\,],N,T\text{-}V_N$), se $N,T > 0$

  total de maneiras sem usar a última moeda, e total com ela
  - 1, se $T = 0$
  - 0, se $T < 0$
  - 0, se $N = 0$

- Exemplo
  - Total({1,5,10}, 3, 35) =Total({1,5}, 2, 35) + Total({1,5,10}, 3, 25)
  - Devolver os 35 em moedas de 1 e 5, ou usar a de 10 e devolver os 25 restantes

Counting Change – código programação dinâmica

```
long tab[MAXVALOR+1]; //versão bottom-up

long contar(int moeda[],int nmoedas,int troco) {
   int i,j,valormoeda;

   tab[0] = 1;
   for(int i=1;i<=troco;i++)
      tab[i] = 0;

   for(i=0; i<nmoedas; i++) {
      valormoeda = moeda[i];
      for (j=valormoeda; j<=troco; j++)
         tab[j] += tab[j-valormoeda];
   }
   return tab[troco];
}
```

Counting Change – exemplo

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Tab	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Com a moeda de valor 1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Tab	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

+ + + +

Counting Change – exemplo

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Tab	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Com a moeda de valor 1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Tab	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

- Com a moeda de valor 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Tab	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3	4	4	4	4	4	5

+ + +

Counting Change – exemplo

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Tab	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Com a moeda de valor 1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Tab	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

- Com a moeda de valor 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Tab	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3	4	4	4	4	4	5

- Com a moeda de valor 10

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Tab	1	1	1	1	1	2	2	2	2	2	4	4	4	4	4	6	6	6	6	6	9

+ + +

Longest Common Subsequence

- Dados
  - dois strings
- Objetivo
  - achar a maior subseqüência comum
- Utilização
  - Seqüenciamento de DNA
  - Busca em texto
- Exemplos
  - ACGTTGCCAG e CATGTGATT

Longest Common Subsequence

- Dados
  - dois strings
- Objetivo
  - achar a maior subseqüência comum
- Utilização
  - Seqüenciamento de DNA
  - Busca em texto
- Exemplos
  - **A**CG**TTG**CC**A**G e C**AT**G**TGA**TT
    - A maior tem tamanho 5: ATTGA (entre outras)
  - Republicano e Democrata

Longest Common Subsequence

- Dados
  - dois strings
- Objetivo
  - achar a maior subseqüência comum
- Utilização
  - Seqüenciamento de DNA
  - Busca em texto
- Exemplos
  - **A**CG**TTG**CC**A**G e C**ATG**T**GA**TT
    - A maior tem tamanho 5: ATTGA (entre outras)
  - R**e**publi**ca**no e D**e**mo**c**r**a**ta
    - A maior tem tamanho 3: eca

Longest Common Subsequence

- Idéia top-down
  - Se o último caracter é igual
    - Então é 1 + a LCS dos strings sem esse caracter
  - Se o último caracter não é igual
    - Calcular a LCS do string s1 inteiro e s2 sem o último caracter
    - Calcular a LCS do string s1 sem o último caracter e s2 inteiro
    - A solução é o que for maior
- Exemplo
  - LCS (“ACTGTT”, “CATGT”) = 1 + LCS (“ACTGT”, “CATG”)
  - LCS (“ACTGT”, “CATG”) = max (LCS (“ACTG”, “CATG”),
    LCS (“ACTGT”, “CAT”))
- A versão bottom-up faz o caminho inverso

Longest Common Subsequence – código

```
int tab[MAX][MAX];

int LCS(char *X, char *Y) {
  int i,j, m=strlen(X), n=strlen(Y);

  for (i=1;i<=m;i++)
    tab[i][0]=0;
  for (j=0;j<=n;j++)
    tab[0][j]=0;
  ...
```

Longest Common Subsequence – código

```
  ...
  for (i=1;i<=m;i++)
    for (j=1;j<=n;j++) {
      if (X[i-1]==Y[j-1])
        tab[i][j]=tab[i-1][j-1]+1;
      else if (tab[i-1][j]>=tab[i][j-1])
        tab[i][j]=tab[i-1][j];
      else
        tab[i][j]=tab[i][j-1];
    }

  return tab[m][n];
}
```

Longest Common Subsequence – exemplo

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

observação: na ilustração os strings estão uma posição à frente

Longest Common Subsequence – imprimir seqüência

```
for (i=1;i<=m;i++)
  for (j=1;j<=n;j++) {
    if (X[i-1]==Y[j-1]) {
      tab[i][j]=tab[i-1][j-1]+1;
      c[i][j] = OK;
    }
    else if (tab[i-1][j]>=tab[i][j-1]) {
      tab[i][j]=tab[i-1][j];
      c[i][j] = CIMA;
    }
    else {
      tab[i][j]=tab[i][j-1];
      c[i][j] = ESQUERDA;
    }
  }
return tab[m][n];
}
```

Longest Common Subsequence – imprimir seqüência

	A	C	G	T	T	G	C	C	A	G	
C	0	0	0	0	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1	1	1	1	1
T	0	1	1	1	1	1	1	1	1	2	2
G	0	1	1	1	2	2	2	2	2	2	2
T	0	1	1	2	2	2	3	3	3	3	3
G	0	1	1	2	3	3	3	3	3	3	3
A	0	1	1	2	3	3	4	4	4	4	4
T	0	1	1	2	3	3	4	4	4	5	5
T	0	1	1	2	3	4	4	4	4	5	5
	0	1	1	2	3	4	4	4	4	5	**5**

Longest Common Subsequence – imprimir seqüência

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

Longest Common Subsequence – imprimir seqüência

```
void imprimeLCS(char *X, char *Y, int i,int j) {
  if (i==0 || j==0)
    return;

  if (c[i][j]==OK) {
    imprimeLCS(X,Y,i-1,j-1);
    cout << X[i-1];
  }
  else if (c[i][j]==CIMA)
    imprimeLCS(X,Y,i-1,j);
  else
    imprimeLCS(X,Y,i,j-1);
}
```

Edit Distance

- Dados
  - dois strings
- Objetivo
  - Achar o número mínimo de operações (inserção, retirada ou substituição de caracteres) para transformar um no outro
- Exemplo
  - ACGGTA e CGATAC
  - Distância 3: ACGGTA → –CGGTA → CGATA → CGATAC
- Aplicações
  - Comparação de arquivos: comando *diff*
  - Correção automática de texto
  - Biologia molecular
  - ...

Edit Distance – recursão

- Seja `dist(s1,s2)` a distância de edição de `s1` e `s2`, ou seja, o mínimo de transformações para `s1` se tornar `s2`
- Passo base (condição de parada):
- `dist("", "") = 0`
- `dist("", s) = strlen(s)`
  - Distância é o tamanho do string, são |s| inserções
- `dist(s, "") = strlen(s)`
  - Distância é o tamanho do string, são |s| retiradas

Edit Distance – recursão

- Passo recursivo para `dist(s1+c1, s2+c2)`
- A distância de edição é a menor entre
  - Substituição: transformar `s1` em `s2` e substituir `c1` por `c2`

```
if (c1==c2)
   dist(s1, s2)
else
   dist(s1, s2) + 1
```

  - Retirada: retirar `c1` e transformar `s1` em `s2+c2`

```
dist(s1, s2+c2) + 1
```

  - Inserção: transformar `s1+c1` em `s2`, e inserir `c2`

```
dist(s1+c1, s2) + 1
```

Edit Distance – recursão

- dist("", "") = 0 *// strings vazios*
- dist("", s) = dist(s, "") = |s| *// tamanho de s*
- dist(s1+c1, s2+c2) = min(

```
        dist(s1    , s2    ) + (c1==c2 ? 0 : 1),   // substituição
        dist(s1    , s2+c2) + 1,   // retirada
        dist(s1+c1, s2    ) + 1)   // inserção
```

- Note que a recursão seria chamada 3 vezes!

Edit Distance – código programação dinâmica

```
tab[0][0] = 0;
for (i=1; i<strlen(s1); i++)
  tab[i][0] = i;
for (j=1; j<strlen(s2); j++)
  tab[0][j] = j;

for (i=0; i<strlen(s1); i++)
  for (j=0; j<strlen(s2); j++) {
    val = (s1[i] == s2[j]) ? 0 : 1;
    tab[i][j] = min( tab[i-1][j-1] + val,
                     tab[i-1][j]+1,
                     tab[i][j-1]+1);
  }
```

Edit Distance – exemplo

		A	C	G	G	T	A
	0	1	2	3	4	5	6
C	1	1	1	2	3	4	5
G	2	2	2	1	2	3	4
A	3	2	3	2	2	3	3
T	4	3	3	3	3	2	3
A	5	4	4	4	4	3	2
C	6	5	4	5	5	4	3

		A	C	G	G	T	A
	.	**R**	R	R	R	R	R
C	I	S	**=**	R	R	R	R
G	I	S	S	**=**	R	R	R
A	I	=	R	I	**S**	R	=
T	I	I	S	I	S	**=**	R
A	I	=	S	S	S	I	**=**
C	I	I	=	R	S	I	**I**

tabela auxiliar para recuperar caminho

Longest Increasing Subsequence

- Dados
  - Uma seqüência de valores
- Objetivo
  - Encontrar a maior subseqüência com valores crescentes
- Exemplos
  - 5 2 4 7 5 3 8 7 9 4 2 1

Longest Increasing Subsequence

- Dados
  - Uma seqüência de valores
- Objetivo
  - Encontrar a maior subseqüência com valores crescentes
- Exemplos
  - 5 **2** **4** 7 **5** 3 **8** 7 **9** 4 2 1
    - Maior subseqüência tem tamanho 5
- Solução por LCS
  - Maior subseqüência comum com 1 2 3 4 5 6 7 8 9 ...
  - Mas dessa forma só seria interessante se a faixa de valores fosse pequena

Longest Increasing Subsequence – código

```
int tam[MAX];              //tamanho da maior (todos = 1)
int predecessor[MAX];//quem vem antes (todos = -1)

int lis(int valor[], int n)
{
  for(int i=0; i<n-1; i++)
    for(int j=i+1; j<n; j++)
      if (valor[j] > valor[i])       //se for maior
        if (tam[i] + 1 > tam[j]) { //e aumentar a
          tam[j] = tam[i] + 1;      //sequencia
          predecessor[j] = i;
        }
  //retornar a posição do maior valor do vetor tam
}
```

Longest Increasing Subsequence – exemplo

Maior subseqüência crescente tem tamanho 5

	0	1	2	3	4	5	6	7	8	9	10	11
valor	5	2	4	7	5	3	8	7	9	4	2	1
tam	1	1	2	3	3	2	4	4	5	3	1	1
predec	-1	-1	1	2	2	1	3	4	6	5	-1	-1

Longest Increasing Subsequence – exemplo

Maior subseqüência crescente tem tamanho 5

	0	1	2	3	4	5	6	7	8	9	10	11
valor	5	2	4	7	5	3	8	7	9	4	2	1
tam	1	1	2	3	3	2	4	4	5	3	1	1
predec	-1	-1	1	2	2	1	3	4	6	5	-1	-1

A seqüência pode ser recuperada pelo `predecessor`

	0	1	2	3	4	5	6	7	8	9	10	11
valor	5	2	4	7	5	3	8	7	9	4	2	1
predec	-1	-1	1	2	2	1	3	4	6	5	-1	-1

Zero-One Knapsack

- Dados
  - Uma lista de *N* itens, cada um com valor V_i e peso W_i,
  - Uma mochila de capacidade *CAP*
- Objetivo
  - determinar o valor máximo de itens que pode ser carregado
  - Cada item pode ser levado ou não (0-1)
- Exemplo
  - Capacidade: 17
  - Peso: `5 7 3 4 8`
  - Valor: `11 14 6 9 17`

Zero-One Knapsack

- Dados
  - Uma lista de N itens, cada um com valor V_i e peso W_i,
  - Uma mochila de capacidade *CAP*
- Objetivo
  - determinar o valor máximo de itens que pode ser carregado
  - Cada item pode ser levado ou não (0-1)
- Exemplo
  - Capacidade: 17
  - Peso: `5   7   3   4   8`
  - Valor: `11  14  6   9  17` – Total: 37

Zero-One Knapsack – recursão

- Seja $C[\,i\,][\,j\,]$ o valor máximo considerando itens 1 até i numa mochila de capacidade j
- Se $i == 0$ ou $j == 0$ *// se não há itens ou não cabe nada*
  - $C[\,i\,][\,j\,] = 0$
- Se $w_i > j$ *// se item i não cabe nessa mochila*
  - $C[\,i\,][\,j\,] = C[\,i-1\,][\,j\,]$ *// considera somente os itens anteriores*
- Se $w_i <= j$
  - $C[\,i\,][\,j\,] = \max \{\, C[\,i-1\,][\,j\,],$ *// mochila sem o item*

    $C[\,i-1\,][\,j-w_i\,] + v_i \,\}$ *// mochila com o item*
- Solução
  - $C[\,N\,][\,CAP\,]$

Zero-One Knapsack – código programação dinâmica

```
int tab[MAXITENS+1][MAXCAPACIDADE+1];

int mochila01(int valor[], int peso[],
              int nitens, int capacidade)
{
  int i,j;

  for (i=0;i<=nitens;i++)
     tab[i][0] = 0;     //mochila sem capacidade

  for (j=0;j<=capacidade;j++)
     tab[0][j] = 0;     //mochila sem itens
     ...
```

Zero-One Knapsack – código programação dinâmica

```
...            //OBS.: o item i está na posicao i-1
for (i=1; i<=nitens; i++)
 for (j=1; j<=capacidade; j++)
  if (peso[i-1] > j)           //se nao cabe
    tab[i][j] = tab[i-1][j];//mochila sem ele
  else
    tab[i][j] = max (  //se cabe, é o máximo da
        tab[i-1][j],    //mochila sem ele
        tab[i-1][j-peso[i-1]]+valor[i-1]
        );              //ou com ele

return tab[nitens][capacidade];
}
```

Zero-One Knapsack – exemplo

- Resultado

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	**0**	**0**	**0**	**0**	**0**	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**
2	**0**	**0**	**0**	**0**	**0**	**11**	**11**	**14**	**14**	**14**	**14**	**14**	**25**	**25**	**25**	**25**	**25**	**25**
3	**0**	**0**	**0**	**6**	**6**	**11**	**11**	**14**	**17**	**17**	**20**	**20**	**25**	**25**	**25**	**31**	**31**	**31**
4	**0**	**0**	**0**	**6**	**9**	**11**	**11**	**15**	**17**	**20**	**20**	**23**	**26**	**26**	**29**	**31**	**34**	**34**
5	**0**	**0**	**0**	**6**	**9**	**11**	**11**	**15**	**17**	**20**	**20**	**23**	**26**	**28**	**29**	**32**	**34**	**37**

Knapsack – podendo repetir item

```
   ...          //OBS.: o item i está na posicao i-1
   for (i=1; i<=nitens; i++)
    for (j=1; j<=capacidade; j++)
     if (peso[i-1] > j)          //se nao cabe
       tab[i][j] = tab[i-1][j];//mochila sem ele
     else
       tab[i][j] = max (  //se cabe, é o máximo da
          tab[i-1][j],    //mochila sem ele
          tab[i-1][j-peso[i-1]]+valor[i-1]
          );              //ou com ele

   return tab[nitens][capacidade];
}
```

Knapsack – exemplo

- Resultado (os itens devem ser ordenados pelo peso)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	**0**	**0**	**0**	**6**	**6**	**6**	**12**	**12**	**12**	**18**	**18**	**18**	**24**	**24**	**24**	**30**	**30**	**30**
2	**0**	**0**	**0**	**6**	**9**	**9**	**12**	**15**	**18**	**18**	**21**	**24**	**27**	**27**	**30**	**33**	**36**	**36**
3	**0**	**0**	**0**	**6**	**9**	**11**	**12**	**15**	**18**	**20**	**22**	**24**	**27**	**29**	**31**	**33**	**36**	**38**
4	**0**	**0**	**0**	**6**	**9**	**11**	**12**	**15**	**18**	**20**	**22**	**24**	**27**	**29**	**31**	**33**	**36**	**38**
5	**0**	**0**	**0**	**6**	**9**	**11**	**12**	**15**	**18**	**20**	**22**	**24**	**27**	**29**	**31**	**33**	**36**	**38**

- Na verdade, é possível fazer tudo em um vetor (sem usar a matriz)

Maximum Interval Sum

- Dados
  - Um seqüência de inteiros quaisquer
- Objetivo
  - Encontrar a seqüência consecutiva de máxima soma
- Exemplo
  - 3 -6 5 4 -3 5 -7 -2 3 -8 7 2

Maximum Interval Sum

- Dados
  - Um seqüência de inteiros quaisquer
- Objetivo
  - Encontrar a seqüência consecutiva de máxima soma
- Exemplo
  - 3 -6 **5 4 -3 5** -7 -2 3 -8 7 2
  - Soma máxima: 11

Maximum Interval Sum – idéia PD

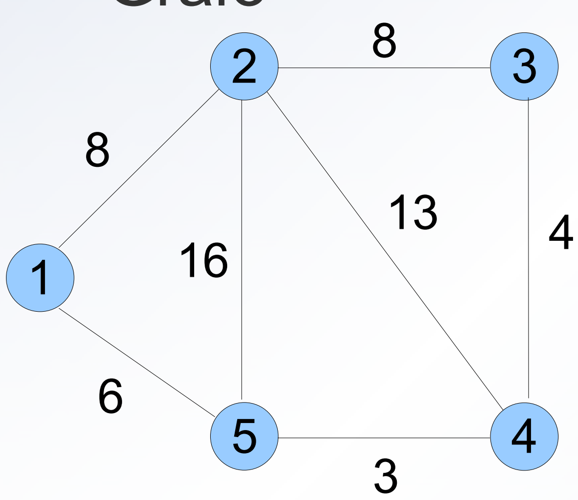
- Acumular a soma com os anteriores
- Recomeçar se a soma anterior for negativa

```
         0   1   2   3   4   5   6   7   8   9  10  11
lista    3  -6   5   4  -3   5  -7  -2   3  -8   7   2
soma     3  -3   5   9   6  11   4   2   5  -3   7   9
             ^                               ^

^ onde parou e recomeçou em seguida
```

All Shortest Paths

- Dados
  - Um grafo com *N* vértices
  - A distância direta entre cada par de vértices
- Objetivo
  - Encontrar o menor caminho entre cada par de vértices
- Exemplo
  - Grafo



Distância direta

	1	2	3	4	5
1	0	8	-	-	6
2	8	0	8	13	16
3	-	8	0	4	-
4	-	13	4	0	3
5	6	16	-	3	0

Menor distância

	1	2	3	4	5
1	0	8	13	9	6
2	8	0	8	12	14
3	13	8	0	4	7
4	9	12	4	0	3
5	6	14	7	3	0

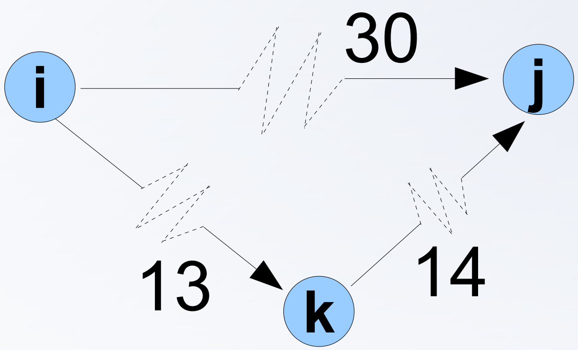
All Shortest Paths – funcionamento do algoritmo Floyd

- Considere D[i][j] para um dado *k*-1
  - A menor distância entre *i* e *j* passando apenas por vértices de índice 1 ... *k*-1
- Desta forma
  - D[i][k] será a menor distância de *i* até *k*, passando apenas por vértices de índice 1 ... *k*-1
  - D[k][j] será a menor distância de *k* até *j*, passando apenas por vértices de índice 1 ... *k*-1
  - Logo, D[i][k] + D[k][j] será a menor distância de *i* até *j*, passando apenas por vértices de índice 1 ... *k*-1, e passando obrigatoriamente por k
- Se D[i][k] + D[k][j] < D[i][j] então é melhor passar por *k*, senão continua com o caminho anterior
- Para encontrar todos os caminhos, basta fazer *k* = 1...*N*

All Shortest Paths – código (algoritmo de Floyd)

- Verifica se é melhor passar pelo *k* para ir de *i* até *j*

```
for(k=1;k<=nv;k++)
   for(i=1;i<=nv;i++)
      for(j=1;j<=nv;j++) {
         val = d[i][k] + d[k][j];
         if (val < d[i][j])
            d[i][j] = val;
      }
```

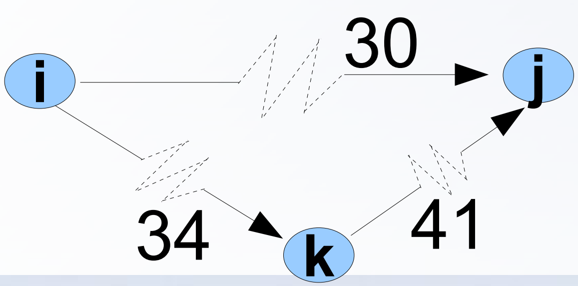


All Shortest Paths – adaptação

- Variação para caminho com maior aresta mínima

```
for(k=1;k<=nv;k++)
   for(i=1;i<=nv;i++)
      for(j=1;j<=nv;j++) {
         val = min(d[i][k],d[k][j]);
         if (val > d[i][j])
            d[i][j] = val;
      }
```

- Exemplo: valor da aresta representa a altura dos túneis
  - d[i][j] será a altura do maior caminhão que pode ir de *i* a *j*

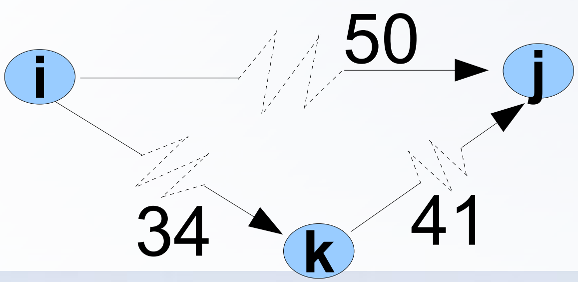


All Shortest Paths – adaptação

- Variação para caminho com menor aresta máxima

```
for(k=1;k<=nv;k++)
   for(i=1;i<=nv;i++)
      for(j=1;j<=nv;j++) {
         val = max(d[i][k],d[k][j]);
         if (val < d[i][j])
            d[i][j] = val;
      }
```

- Exemplo: valor da aresta representa tempo de vôo
  - d[i][j] será mínimo vôo mais longo para ir de *i* a *j*



Multiplicação de Cadeia de Matrizes

- Dados
  - Matrizes A_1, A_2, ..., A_n, cada A_i de dimensão $P_{i-1} \times P_i$
- Objetivo
  - Colocar parênteses em $A_1 \times A_2 \times \ldots \times A_n$ para minimizar o número de multiplicações escalares
- Exemplo
  - Seja A_1 uma matriz 10×100, A_2 100×5 e A_3 5×50
  - $((A_1 . (A_2 . A_3))$
  - $((A_1 . A_2) . A_3))$

Multiplicação de Cadeia de Matrizes

- Dados
  - Matrizes $A_1, A_2, ..., A_n$, cada A_i de dimensão $P_{i-1} \times P_i$
- Objetivo
  - Colocar parênteses em $A_1 \times A_2 \times ... \times A_n$ para minimizar o número de multiplicações escalares
- Exemplo
  - Seja A_1 uma matriz 10 x 100, A_2 100 x 5 e A_3 5 x 50
  - $((A_1 . (A_2 . A_3))$ – 100 x 5 x 50 + 10 x 100 x 50 = 75000
  - $((A_1 . A_2) . A_3))$ – 10 x 100 x 5 + 10 x 5 x 50 = 7500
  - A segunda forma faz 10 vezes menos!

Multiplicação de Cadeia de Matrizes – recursão

- M[i][j] = mínimo de multiplicações escalares de $A_i \ldots A_j$
- Se i == j
  - M[i][j] = 0
- Se i < j
  - $M[\,i\,][\,j\,] = \min \{\, M[\,i\,][\,k\,] + M[\,k+1\,][\,j\,] + P_{i-1}P_kP_j \,\}$
  - para todo k, $i \leq k < j$

Referências

- Programming Challenges
  - www.programming-challenges.com
- World of Seven
  - http://www.comp.nus.edu.sg/~stevenha/myteaching/
- Online-judge Universidad Valladolid
  - http://acm.uva.es/problemset/