Aula 13:
Programação Dinâmica

Disciplina: Maratona de Programação 1

Profs. Edmilson Marmo e Luiz Olmes
edmarmo@unifei.edu.br, olmes@unifei.edu.br

UNIFEI
UNIVERSIDADE FEDERAL DE ITAJUBÁ

Nas aulas anteriores...

- **O QUE JÁ ESTUDAMOS?**
- Introdução à Maratona
- Problemas ad hoc
- Standard Template Library (STL)
- Grafos: DFS e BFS
- Grafos: algoritmos de Dijkstra e Tarjan
- Union-Find

- **OBJETIVOS:**
- Entender o conceito de Programação Dinâmica.
- Identificar situações de uso.
- Analisar formas de implementação.

Introdução

- Uma importante parte dos problemas em uma competição pode ser resolvida através de Programação Dinâmica (também chamada de PD).

- PD é uma técnica de **projeto de algoritmo** que pode ser usada para encontrar soluções ótimas para problemas e contar o número de soluções.

- É uma técnica normalmente baseada em uma **fórmula de recursão** e **alguns estados iniciais**.

Introdução

- A solução de um problema é construída a partir de outras subsoluções computadas anteriormente.

- Soluções com PD possuem **complexidade polinomial**, garantindo um desempenho muito melhor do que soluções “diretas” como *backtracking* e força-bruta, que podem executar em **tempo exponencial**.

- Um algoritmo que utilize PD **resolve cada subproblema uma única vez:**
  - **Grava seu resultado em uma tabela**, de maneira que não é necessário resolvê-lo novamente toda vez que houver uma instância idêntica daquele subproblema.

Introdução

- **Em resumo**:
  - PD é um nome interessante para recursão utilizando uma tabela.
  - Ao invés de resolver subproblemas recursivamente, procura-se resolvê-los sequencialmente, guardando suas soluções em uma tabela (um vetor ou matriz).
- A questão principal é **resolver os problemas na ordem correta**. Assim, sempre que uma solução para um subproblema é necessária, ela já se encontrará na tabela.
- O **desempenho** do algoritmo será, em geral, **proporcional ao tamanho da tabela** (liner para um vetor, quadrático para uma matriz, etc.)

Introdução

- O termo **“programação”** em **Programção Dinâmica** tem muito pouco a ver com escrever código.
    - Foi cunhado por Richard Bellman nos anos 50, quando a programação de computadores era algo tão “místico” e “esotérico” que mal tinha um nome.

- Naquela época, programação significava **“planejamento”**.
    - Nesse contexto, **“Programação”** refere-se à construção da tabela que armazena as soluções das subinstâncias, o planejamento **“otimizado”** para um processo multiestágio.

Introdução

- **PLANEJAR:**
  - Encontre uma subestrutura ótima para o problema.
  - Busque uma forma de resolvê-lo a partir de instâncias menores do mesmo subproblema (**fórmula de recorrência**).

- **ELIMINE A REDUNDÂNCIA:**
  - Abordagens “diretas” poderiam recalcular a resposta para um mesmo subproblema dezenas (até milhares) de vezes. É preciso evitar este retrabalho.

Introdução

- Existem dois principais métodos para aplicar PD:
- **Memoization** (*Top-Down*):
  - Utiliza recursão para resolver o problema e armazena os resultados dos subproblemas em uma tabela;
  - Quando um subproblema é encontrado novamente, o resultado armazenado é usado em vez de recalcular.
- **Tabulation** (*Bottom-up*):
  - Constrói a solução para o problema de forma iterativa, partindo dos subproblemas mais simples até chegar ao problema original.
  - Os resultados intermediários são armazenados em uma tabela.

Como eliminar o retrabalho?

- Um exemplo bastante conhecido: **Fibonacci!**

- Fibonacci é um problema claramente recursivo e sua implementação mais direta seria:

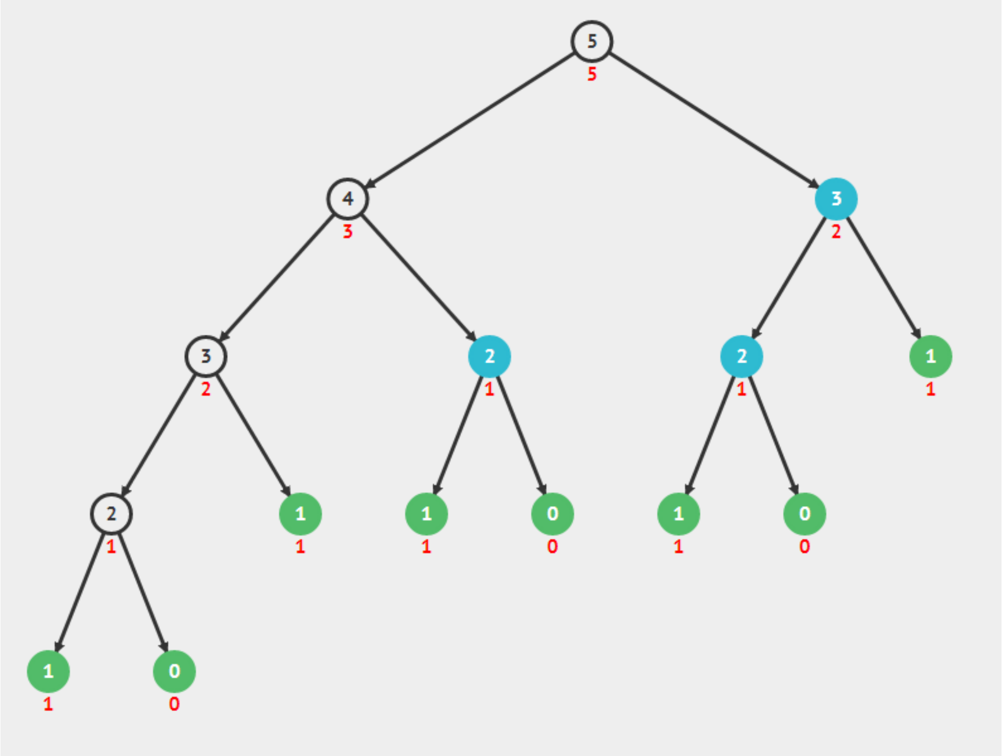
```
int fiboRecursivo (int n)
{
   if (n <= 1)
     return n;
   return fiboRecursivo (n-1) + fiboRecursivo (n-2);
}
```

Como eliminar o retrabalho?

- O algoritmo recursivo, apesar de resolver corretamente o problema, possui **complexidade no tempo bastante indesejável**.

- Cada chamada da função dispara outras duas chamadas. O tempo de execução do algoritmo aumenta exponencialmente com a entrada *n*, tornando-se **$O(2^n)$**.

Como eliminar o retrabalho?



Como eliminar o retrabalho?

Memoization:

- Utiliza-se um vetor, por exemplo **`memo[MAX]`**, que funciona como *cache* para armazenar os resultados já calculados das chamadas recursivas.
- O valor de Fibonacci é calculado apenas uma vez, tempo linear **O(n)**, visto que cada subproblema é resolvido uma única vez.

```
int fiboMemoPD (int n)
{
   if (n <= 1) return n;
   if (memo[n] != -1)
     return memo[n];
   memo[n] = fiboMemoPD(n-1) + fiboMemoPD(n-2);
   return memo[n];
}
```

Como eliminar o retrabalho?

Tabulation:

- Os resultados **intermediários também são armazenados** em um vetor.
- O evento mais “custoso” da função é um *loop* interno, que executa apenas **n-1** vezes (proporcional ao tamanho da tabela). Tempo linear: **O(n)**.

```
int fiboPD (int n)
{
   if (n <= 1) return n;
   int vet[MAX];
   vet[0] = 0; vet[1] = 1;

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

O problema das moedas

- Suponha que seja dado um conjunto de valores representando moedas e uma quantia alvo de dinheiro *n*:
  - `coins` = { c_1, c_2, … , c_k }

  O objetivo será: **construir a quantidade *n* utilizando o mínimo de moedas**.

- Por exemplo, se `coins={1,2,5}` e n = 12, a solução ótima é 5+5+2=12.

- Neste caso, a **solução "gulosa"** parece resolver bem o problema: escolher sempre a maior moeda possível.

- **Esta estratégia é sempre ótima?**

O problema das moedas

- Suponha agora que o conjunto de moedas seja: `coins={1,3,4}` e $n = 6$.

O problema das moedas

- Suponha agora que o conjunto de moedas seja: `coins={1,3,4}` e $n = 6$.

- **Qual seria a solução ótima?**

O problema das moedas

- Suponha agora que o conjunto de moedas seja: `coins={1,3,4}` e $n = 6$.

- **Qual seria a solução ótima?**

- A solução ótima tem apenas **duas moedas (3 + 3 = 6)**, mas a **estratégia gulosa** produz uma solução com **três moedas (4 + 1 + 1 = 6)**.

- Usando PD, pode-se criar um algoritmo que é semelhante a uma solução de força bruta, mas também é eficiente.

O problema das moedas

- Para usar PD, deve-se **formular o problema recursivamente** para que a solução possa ser calculada a partir de soluções para subproblemas menores.

- Uma solução naturalmente recursiva é calcular a função **`solve(x)`** respondendo a pergunta:
  - Qual é o número mínimo de moedas necessárias para formar uma ***soma x***?

O problema das moedas

- Usando o mesmo exemplo anterior: `coins={1,3,4}` e $n = 6$.

solve(0) = 0

solve(1) = 1

solve(2) = 2

solve(3) = 1

solve(4) = 1

solve(5) = 2

solve(6) = 2

...

O problema das moedas

- A propriedade essencial de `solve(x)` é que seus valores podem ser recursivamente calculados a partir de seus valores menores.

- Por exemplo, `solve(10)=3`, pois somente 3 moedas são necessárias:

  3 + 3 + 4.

- A ideia é focar na primeira moeda que é escolhida para a soma.

- No exemplo, a primeira moeda pode ser 1, 3 ou 4.

O problema das moedas

- Se a primeira moeda escolhida for 1, a tarefa restante é formar a soma 9 usando o número mínimo de moedas, que **é um subproblema** do problema original.

- O mesmo se aplica às moedas 3 e 4.

O problema das moedas

- Se a primeira moeda escolhida for 1, a tarefa restante é formar a soma 9 usando o número mínimo de moedas, que **é um subproblema** do problema original.

- O mesmo se aplica às moedas 3 e 4.

- **Portanto:**

```
solve(x) = min(solve(x − 1) + 1,
               solve(x − 3) + 1,
               solve(x − 4) + 1)
```

O problema das moedas

- O caso base da recursão é **`solve(0)=0`**, porque nenhuma moeda é necessário para forma uma soma de 0.

- Observe que:

  `solve(10) = solve(7) + 1 = solve(4) + 2 = solve(0) + 3 = 3.`

O problema das moedas

- Agora é possível gerar uma relação de recorrência para este caso:

$$\texttt{solve}(x) = \begin{cases} \infty & x < 0 \\ 0 & x = 0 \\ \min_{c \in \texttt{coins}} \texttt{solve}(x - c) + 1 & x > 0 \end{cases}$$

```
int solve (int x)
{
   if (x < 0) return INF;
   if (x == 0) return 0;
   int best = INF;
   for (auto c : coins) {
     best = min (best, solve(x-c)+1);
   }
   return best;
}
```

O problema das moedas

- **Memoization:**

```
bool ready[MAX];
int value[MAX];

int solve (int x)
{
   if (x < 0) return INF;
   if (x == 0) return 0;
   if (read[x]) return value[x];
   int best = INF;
   for (auto c : coins) {
     best = min (best, solve(x-c)+1);
   }
   read[x] = true; value[x] = best;
   return best;
}
```

O problema das moedas

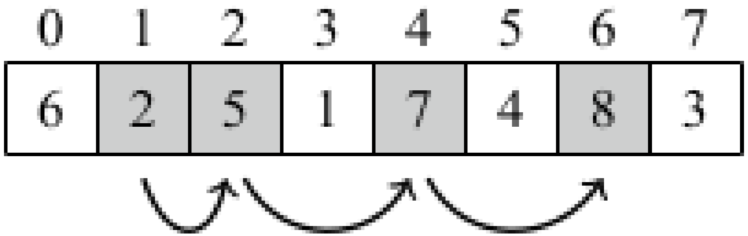
- **Tabulation:**

```
int solve (int x)
{
  value[0] = 0;
  for (int i=1; i <= x; i++) {
    value[i]=INF;
    for (auto c : coins) {
      if (i-c >= 0) {
        value[i] = min (value[i], value[i-c]+1);
      }
    }
  }
  return value[x];
}
```

O problema da Subsequência Crescente mais Longa

- A ***longest increasing subsequence*** em um vetor de *n* elementos é uma sequência de comprimento máximo de elementos do vetor que vai da esquerda para a direita, e cada elemento na sequência é maior que o elemento anterior.

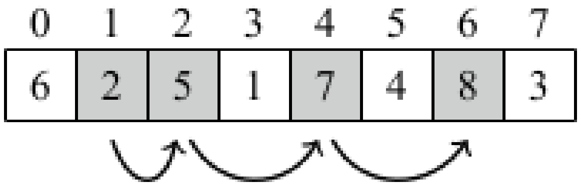
- A ilustração a seguir mostra a subsequência crescente mais longa do vetor:



O problema da Subsequência Crescente mais Longa

- Seja `length(k)` o comprimento da maior subquência crescente que termina na posição k.
- Se calcularmos todos os valores de **`length(k)`**, teremos o comprimento da maior subsequência crescente.

length(0)=1
length(1)=1
length(2)=2
length(3)=1
length(4)=3
length(5)=2
length(6)=4
length(7)=2



O problema da Subsequência Crescente mais Longa

- Para calcular o valor de **`length(k)`**, devemos encontrar uma posição $i < k$ tal que:

  `array[i] < array[k]` e **`length(i)`** seja o maior possível.

  Sabemos que **`length(k)=length(i)+1`**, porque é a maneira ótima de anexar **`array[k]`** a uma subsequência.

- Se não existir tal posição *i*, então, **`length(k)=1`**, o que significa que a subsequência contém apenas **`array[k]`**.

O problema da Subsequência Crescente mais Longa

- Como todos os valores da função podem ser calculados a partir de seus valores menores, podemos usar PD para calcular os valores menores.

O problema da Subsequência Crescente mais Longa

- Como todos os valores da função podem ser calculados a partir de seus valores menores, podemos usar PD para calcular os valores menores.

- No código abaixo, os valores da função serão armazenados no vetor **`length`**.

```
for (int k=0; k<n; k++) {
  length[k] = 1;
  for (int i=0; i<k; i++) {
    if (array[i] < array[k]) {
      length[k] = max (length[k], lenght[i]+1);
    }
  }
}
```

Caminhos em uma Matriz

- O objetivo deste problema é **encontrar um caminho do canto superior esquerdo para o canto inferior direito** de uma matriz $n \times n$.
- A restrição é de que **só é possível mover apenas para baixo e para direita**.
- Cada célula contém um inteiro, e o caminho deve ser construído de modo que a soma dos valores ao longo do caminho seja o maior possível.

3	7	9	2	7
9	8	3	5	5
1	7	9	8	5
3	8	6	4	10
6	3	9	7	8

Caminhos em uma Matriz

- Suponha que as linhas e as colunas da matriz sejam numeradas de 1 a n e que **`value[y][x]`** seja igual ao valor da **submatriz (y,x)**.

- Seja **`sum(y,x)`** a soma máxima em um caminho do canto superior esquerdo ao canto inferior direito da **submatriz (y,x)**.

- Desta forma, **`sum(n,n)`** terá a soma máxima do canto superior esquerdo ao canto inferior direito.

Caminhos em uma Matriz

- A partir da observação de que um caminho que termina na **submatriz (y, x)** pode vir da **célula (y, x-1)** ou da **célula (y-1, x)**, deve-se selecionar a direção que maximiza a soma.

- Assumimos que **`sum(y,x)=0`** se **y=0** ou **x=0**. Desta forma, a seguinte fórmula recursiva funciona para todos as células:

```
sum(y,x) = max(sum(y,x-1), sum(y-1,x)) + value[y][x]
```

Caminhos em uma Matriz

- Como a função **`sum`** tem dois parâmetros, a matriz de PD também tem duas dimensões:

```
int sum[N][N];
```

- Para calcular a soma:

```
for (int y=1; y<=n; y++) {
  for (int x=0; x<=n; x++) {
     sum[y][x] = max (sum[y][x-1], sum[y-1][x]) + value[y][x];
  }
}
```

O Problema da Distância de Edição

- Distância de Edição (ou *Edit Distance*) mede o número mínimo de operações necessárias para transformar uma *string* s1 em outra *string* s2.

- É um problema amplamente utilizado em áreas como **Processamento de Linguagem Natural** e **Bioinformática**.

- A distância de edição é assim chamada porque pode ser imaginada como o número mínimo de *edições* (inserções, remoções e substituições de caracteres) necessárias para transformar a primeira *string* na segunda.

O Problema da Distância de Edição

- **Exemplo 1**:
  - **Strings:**
    - s1="cat"
    - s2="cut"
  - **Operações necessárias:**
    - Substituir 'a' por 'u'.
  - **Distância de edição: 1**

O Problema da Distância de Edição

- **Exemplo 2**:
  - **Strings:**
    - s1="kitten"
    - s2="sitting"
  - **Operações necessárias:**
    - Substituir 'k' por 's'.
    - Substituir 'e' por 'i'.
    - Inserir 'g' no final.
  - **Distância de edição: 3**

O Problema da Distância de Edição

- Uma medida natural da distância entre duas *strings* é a extensão segundo a qual elas podem ser **alinhadas**, ou **emparelhadas**.

O Problema da Distância de Edição

- Uma medida natural da distância entre duas *strings* é a extensão segundo a qual elas podem ser **alinhadas**, ou **emparelhadas**.

- Um **alinhamento** é uma maneira de escrever as *strings* uma sobre a outra.

- Por exemplo, dois possíveis alinhamentos de **SNOWY** e **SUNNY**

S	**_**	**N**	**O**	**W**	**Y**		**_**	**S**	**N**	**O**	**W**	**_**	**Y**
S	**U**	**N**	**N**	**_**	**Y**		**S**	**U**	**N**	**_**	**_**	**N**	**Y**

Custo : 3

Custo : 5

O Problema da Distância de Edição

- O objetivo é encontrar a **distância de edição** entre duas *strings* `x[1..m]` e `y[1..n]`.

- **O que seria um bom subproblema?**

O Problema da Distância de Edição

- O objetivo é encontrar a **distância de edição** entre duas *strings* `x[1..m]` e `y[1..n]`.

- **O que seria um bom subproblema?**

- Obviamente, ele deveria ser parte do caminho para resolver o problema inteiro.

O Problema da Distância de Edição

- Uma estratégia é observar a distância de edição entre algum ***prefixo*** da primeira *string*, `x[1..i]`, e algum ***prefixo*** da segunda *string*, `y[1..j]`.

O Problema da Distância de Edição

- Uma estratégia é observar a distância de edição entre algum ***prefixo*** da primeira *string*, `x[1..i]`, e algum ***prefixo*** da segunda *string*, `y[1..j]`.

- A distância de edição destes prefixos é o subproblema $E(i, j)$. Portanto, o objetivo final é computar $E(m, n)$.

O Problema da Distância de Edição

- Uma estratégia é observar a distância de edição entre algum ***prefixo*** da primeira *string*, `x[1..i]`, e algum ***prefixo*** da segunda *string*, `y[1..j]`.

- A distância de edição destes prefixos é o subproblema $E(i, j)$. Portanto, o objetivo final é computar $E(m, n)$.

- Para funcionar, é necessário alguma forma de expressar $E(i, j)$ em termos de subproblemas menores.

O Problema da Distância de Edição

- O que sabemos sobre o melhor alinhamento entre `x[1..i]` e `y[1..j]` ?

O Problema da Distância de Edição

- O que sabemos sobre o melhor alinhamento entre `x[1..i]` e `y[1..j]` ?

- Existem **3 situações** para a coluna mais à direita:

`x[i]`	ou	–	ou	`x[i]`
–		`y[j]`		`y[j]`

O Problema da Distância de Edição

- Existem 3 situações para a coluna mais à direita:

`x[i]`	ou	`–`	ou	`x[i]`
`–`		`y[j]`		`y[j]`

- O primeiro caso incorre em um custo de 1 para essa particular coluna e resta alinhar `x[1..i−1]` com `y[1..j]`.
  - Mas isso é exatamente o subproblema $E(i - 1, j)$.

- No segundo caso, também com custo 1, ainda é necessário alinhar `x[1..i]` com `y[1..j−1]`.
  - Isso é novamente um subproblema $E(i, j - 1)$.

O Problema da Distância de Edição

- Existem 3 situações para a coluna mais à direita:

`x[i]`	ou	`–`	ou	`x[i]`
`–`		`y[j]`		`y[j]`

- No terceiro caso, o custo será 1 (se `x[i]` $\neq$ `y[j]`) ou 0 (se `x[i]` = `y[j]`).

- Portanto, é possível expressar $E(i, j)$ em termos de 3 subproblemas menores:
  - $E(i - 1, j)$;
  - $E(i, j - 1)$;
  - $E(i - 1, j - 1)$.

O Problema da Distância de Edição

- Como não é possível prever qual é o melhor, é necessário testar todos os 3 e selecionar o melhor:

$$E(i, j) = \min \{ 1 + E(i-1, j), 1 + E(i, j-1), \text{dif}(i, j) + E(i-1, j-1) \}$$

- Por conveniência: dif (i, j) é definido como 0 (se `x[i]= y[j]`) e 1, caso contrário.

O Problema da Distância de Edição

- Por exemplo, na computação da distância de edição entre EXPONENCIAL e POLINOMIAL, o subproblema $E(4, 3)$ corresponde aos prefixos EXPO e POL.

- A coluna mais à direita de seu melhor alinhamento tem de ser uma das seguintes:

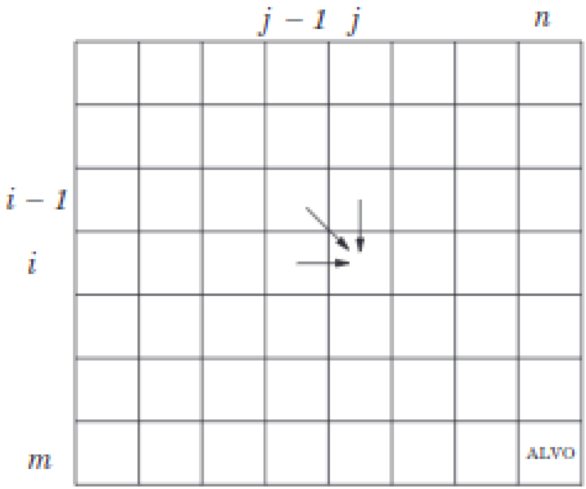
`O`	ou	`–`	ou	`O`
`–`		`L`		`L`

- Assim, $E(4, 3) = \min \{ 1 + E(3, 3), 1 + E(4, 2), 1 + E(3, 2) \}$

O Problema da Distância de Edição

- As respostas para todos os subproblemas $E(i, j)$ formam uma tabela bidimensional.

(a)



(b)

		P	O	L	I	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
C	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

O Problema da Distância de Edição

- Os casos base para preencher $E(i, j)$ são $E(0, \_)$ e $E(\_ , 0)$.

- $E(0, j)$ é a **distância de edição entre o prefixo de x de tamanho 0**, ou seja, a ***string* vazia**, e as primeiras j letras de y, claramente j.

- Similarmente, $E(i, 0) = i$.

O Problema da Distância de Edição

- Código resultante em C:

```
int editDistance(char* s1, char* s2) {
  int m = strlen(s1);
  int n = strlen(s2);
  int PD[m + 1][n + 1];

  // Inicialização da primeira coluna (E(i, 0))
  for (int i = 0; i <= m; i++)
    PD[i][0] = i;

  // Inicialização da primeira linha (E(0, j))
  for (int j = 0; j <= n; j++)
    PD[0][j] = j;
...
```

O Problema da Distância de Edição

- Código resultante em C (**continuação**):

```
...
  // Preenchendo a matriz dp
  for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
      if (s1[i-1] == s2[j-1]) {
        PD[i][j] = PD[i-1][j-1]; // Sem custo extra
      } else {
        PD[i][j] = 1 + min(PD[i-1][j], PD[i][j-1], PD[i-1][j-1]);
      }
    }
  }
  return PD[m][n]; // Resposta final
}
```

O Problema da Distância de Edição

- Função que calcula o menor valor entre os três candidatos:

```
int min (int a, int b, int c) {
  if (a < b && a < c)
    return a;
  if (b < c)
    return b;
  return c;
}
```

O Problema da Mochila

- O termo **mochila** se refere a problemas em que **um conjunto de objetos** é dado, e subconjuntos com **algumas propriedades** precisam ser encontrados.

- Problemas de mochila podem frequentemente ser resolvidos usando programação dinâmica.

Um exemplo:

- Durante um roubo, o ladrão encontra muito mais objetos do que esperava, e tem de decidir o que levar. Sua bolsa (ou "mochila") pode carregar um peso total de no máximo W quilos. Existem n itens entre os quais escolher, de pessos $w_1, \dots, w_n$ e valor em reais $v_1, \dots, v_n$. **Qual a melhor combinação de itens que ele pode colocar na mochila**?

O Problema da Mochila

- Por exemplo, tome $W = 10$ e:

Item	Peso	Valor
1	6	$30
2	3	$14
3	4	$16
4	2	$9

- Há duas versões desse problema. Se, por um lado, existem **quantidades ilimitadas** disponíveis de cada item, a escolha ótima é selecionar o item 1 e dois itens 4 (Total: $48).

O Problema da Mochila

- Por exemplo, tome ***W* = 10** e:

Item	Peso	Valor
1	6	$30
2	3	$14
3	4	$16
4	2	$9

- Por outro lado, se **há apenas um de cada item** (***o ladrão invadiu uma galeria de artes, digamos***), então a mochila ótima contém os itens 1 e 3 (Total: $46).

O Problema da Mochila

Mochila com repetição:

- Como sempre, a principal questão em PD é: **quais são os subproblemas**?

O Problema da Mochila

Mochila com repetição:

- Como sempre, a principal questão em PD é: **quais são os subproblemas**?

- Neste caso, é possível reduzir o problema original de duas maneiras:

  1) Examinar as mochilas de capacidade menor $w \leq W$.

  2) Examinar menos itens (por exemplo, itens 1, 2, … , j para $j \leq n$)

- Normalmente alguma experimentação é necessária para descobrir exatamente qual funciona.

O Problema da Mochila

- A primeira restrição se refere a capacidades menores. De acordo com isso, defina:
  - $K(w)$ = valor máximo alcançável por uma mochila de capacidade w.

- **Como expressar isso em termos de subproblemas menores?**

O Problema da Mochila

- A primeira restrição se refere a capacidades menores. De acordo com isso, defina:
  - $K(w)$ = valor máximo alcançável por uma mochila de capacidade w.

- **Como expressar isso em termos de subproblemas menores?**

- Se a solução ótima para $K(w)$ inclui o item i, então, ao se remover este item da mochila, restará uma solução ótima para $K(w - w_i)$.

- Em outras palavras, **$K(w)$ é simplesmente $K(w - w_i) + v_i$**, para algum i.

O Problema da Mochila

- Como não se sabe qual *i*, deve-se tentar todas as possibilidades:

$$K(w) = \max_{i:wi \leq w} \{ K(w - w_i) + v_i \}$$

- Por convenção, o máximo sobre um conjunto vazio é 0.

- O código desta solução surge naturalmente, como o **preenchimento de uma tabela unidimensional** de comprimento $W + 1$, da esquerda para a direita.

O Problema da Mochila

- Código resultante em C:

```c
int mochilaComRepeticao(int W, int n, int val[], int wt[]) {
  // Vetor para armazenar os valores máximos para cada capacidade
  int PD[W + 1];

  // Inicializando o vetor PD com zeros
  for (int w = 0; w <= W; w++) {
    PD[w] = 0;
  }
...
```

O Problema da Mochila

- Código resultante em C (**continuação**):

```
...
  // Preenchendo o vetor PD
  for (int w = 0; w <= W; w++) {
    for (int i = 0; i < n; i++) {
      if (wt[i] <= w) { // Verifica se o item cabe na mochila
        PD[w] = max(PD[w], PD[w-wt[i]] + val[i]);
      }
    }
  }
  return PD[W];
}
```

O Problema da Mochila

Mochila sem repetição:

- **Como resolver se repetições não são permitidas?**

O Problema da Mochila

Mochila sem repetição:

- **Como resolver se repetições não são permitidas?**

- Saber que o valor $K(w - w_n)$ é muito grande não ajuda, pois não se sabe se o item n já foi usado ou não nesta solução parcial.

- É preciso refinar o conceito de subproblema para contemplar informação adicional sobre os itens em uso.

O Problema da Mochila

- Será necessário um segundo parâmetro, $0 \leq j \leq n$:

$K(w, j)$ = valor máximo alcançável usando uma mochila de capacidade w e itens $1, \ldots, j$.

- **A resposta é**, então, $\boldsymbol{K(W, n)}$.

- Novamente, vem a questão: **como podemos expressar um subproblema $\boldsymbol{K(w, j)}$ em termos de subproblemas menores?**

O Problema da Mochila

- **A resposta é**: ou o item j é necessário para se alcançar o valor ótimo, ou não é:

$$K(w, j) = \max \{ K(w - w_j, j - 1) + v_j, K(w, j - 1) \}.$$

- O algoritmo, portanto, consiste em preencher uma **tabela bidimensional**, com W+1 linhas e n+1 colunas. Cada célula da tabela toma tempo apenas constante, assim, muito embora ela seja muito maior que no caso anterior, o tempo de execução permanece o mesmo **O(*nW*)**.

O Problema da Mochila

- Código resultante em C:

```c
int mochilaSemRepeticao(int W, int n, int val[], int wt[]) {

  // Criação da matriz PD
  int PD[n + 1][W + 1];

...
```

O Problema da Mochila

- Código resultante em C (**continuação**):

```
...
  // Preenchendo a matriz PD
  for (int w = 0; w <= W; w++) {
    for (int i = 0; i < n; i++) {
      if (i == 0 || w == 0) {
        PD[i][w] = 0; // Sem itens ou sem capacidade
      } else if (wt[i-1] <= w) {
        // Escolhe o máximo entre não incluir ou incluir o item
        PD[i][w] = max(PD[i-1][w], PD[i-1][w - wt[i-1]] + val[i-1]);
      } else {
        // Não pode incluir o item
        PD[i][w] = PD[i-1][w];
      }
    }
  }
  return PD[n][W];
}
```

Finalizando...

- **Programação dinâmica não é facil**.

- **Requer muito estudo e prática**.

- Os primeiros problemas demorarão a sair… mas, não desanime.

- Como em todos os desafios da maratona de programação, logo os problemas elementares se tornarão muito fáceis e você estará procurando outros problemas mais difícieis para resolver.

Dúvidas?

Aula 13:
Programação Dinâmica

***Disciplina*: Maratona de Programação 1**

Profs. Edmilson Marmo e Luiz Olmes
edmarmo@unifei.edu.br, olmes@unifei.edu.br

UNIFEI
UNIVERSIDADE FEDERAL DE ITAJUBÁ