Programação Dinâmica

prof. Fábio Luiz Usberti
prof. Cid Carvalho de Souza

MC521 - Desafios de Programação I

Instituto de Computação - UNICAMP - 2018

Sumário

Introdução

- Programação dinâmica (PD) é uma técnica de solução de problemas desenvolvida por Richard E. Bellman (1953).
- Assim como o paradigma de divisão e conquista, resolve problemas combinando soluções de subproblemas.
- Um algoritmo de PD resolve cada subproblema uma única vez, armazenando a resposta em uma tabela e portanto, reduzindo o esforço computacional de recalcular a solução de um subproblema previamente resolvido.

Introdução

- PD é tipicamente aplicado a problemas de otimização. Nesses problemas, temos um espaço de soluções viáveis a ser explorado, onde cada solução possui um valor. O objetivo consiste em encontrar uma solução viável com valor ótimo (máximo ou mínimo).
- Para aplicar PD é desejável que o problema possua duas propriedades:

1. Subestrutura ótima: As soluções ótimas do problema incluem soluções ótimas de subproblemas.
2. Sobreposição de subproblemas: O cálculo da solução através de recursão implica no recálculo de subproblemas.

Sequência de Fibonacci

- A sequência de Fibonacci é uma sequência de números naturais proposta no século XIII pelo matemático Leonardo de Pisa, conhecido por Fibonacci.
- Um número dessa sequência é formado pela soma dos dois números anteriores da mesma sequência, com exceção dos dois primeiros números:

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad 55 \quad 89 \quad 144 \ \ldots$$

- Seja F_n o n-ésimo número da sequência, ou seja:

$$F_0 = 0, \quad F_1 = 1, \quad F_2 = 1, \quad F_3 = 2 \ \ldots$$

- Como projetar um algoritmo de programação dinâmica para determinar F_n?

Algoritmo de PD

- O problema de determinar F_n possui subestrutura ótima?

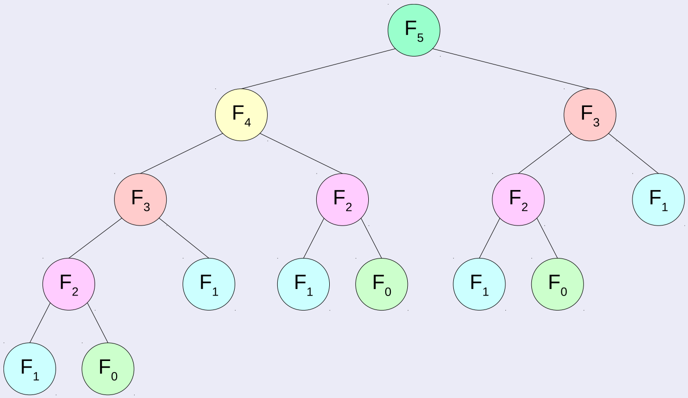
$$F_n = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ F_{n-1} + F_{n-2} & \text{se } n > 1 \end{cases}$$

- A recorrência acima mostra que o problema possui subestrutura ótima, uma vez que o problema F_n pode ser resolvido a partir dos subproblemas F_{n-1} e F_{n-2}.

Sequência de Fibonacci

Algoritmo de PD

- O problema de determinar F_n possui sobreposição de subproblemas?



- A figura acima mostra que um mesmo subproblema pode aparecer mais de uma vez em cada subárvore.

Implementações de PD: **Top-down x Bottom-up**

- Existem duas técnicas para evitar o recálculo de subproblemas:

1. Top-down: Consiste em manter a estrutura recursiva do algoritmo, registrando em uma tabela o valor ótimo para subproblemas já computados e verificando, antes de cada chamada recursiva, se o subproblema a ser resolvido já foi computado.
2. Bottom-up: Consiste em preencher uma tabela que registra o valor ótimo para cada subproblema de forma apropriada, isto é, a computação do valor ótimo de cada subproblema depende somente de subproblemas já previamente computados. Elimina completamente a recursão.

PD: implementação top-down

```
int memo[100];

int fib(int n) {
  if (n < 2) return n;
  else if (memo[n] >= 0) return memo[n];
  return memo[n] = fib(n-1)+fib(n-2);
}

int main() {
  int n;
  scanf("%d", &n);
  memset(memo, -1, sizeof memo);
  printf("fib(%d) = %d\n", n, fib(n));
  return 0;
}
```

PD: implementação bottom-up

```
int fib(int n) {
  int valor[n+1];
  valor[0] = 0;
  valor[1] = 1;

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

int main() {
  int n;
  scanf("%d", &n);
  printf("fib(%d) = %d\n", n, fib(n));
  return 0;
}
```

O Problema da Mochila

Dada uma mochila de capacidade W (inteiro) e um conjunto de n itens com tamanho w_i (inteiro) e valor c_i associado a cada item i, queremos determinar quais itens devem ser colocados na mochila de modo a **maximizar** o valor total armazenado, respeitando sua capacidade.

- Podemos fazer as seguintes suposições:
  - $\sum_{i=1}^{n} w_i > W$;
  - $0 < w_i \leq W$, para todo $i = 1, \ldots, n$.

- Como podemos projetar um algoritmo para resolver o problema?
- Existem 2^n possíveis subconjuntos de itens: um algoritmo de força bruta é impraticável.
- É um problema de otimização. Será que tem subestrutura ótima?
- Considere a possibilidade do item n estar ou não em uma solução ótima:

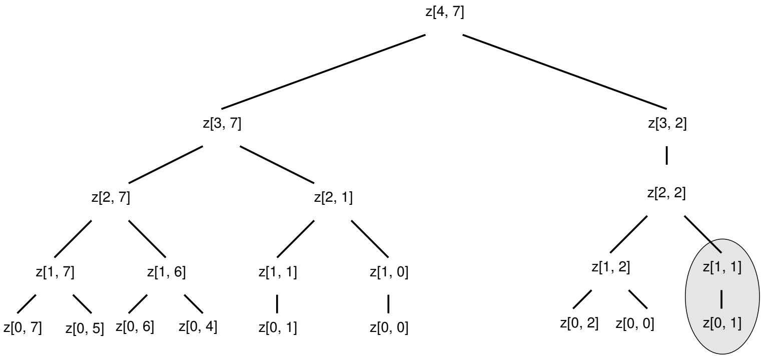
1. Se o item n estiver na solução ótima, o valor desta solução será c_n mais o valor da melhor solução do problema da mochila com capacidade $W - w_n$ considerando-se só os $n - 1$ primeiros itens.
2. Se o item n não estiver na solução ótima, o valor ótimo será dado pelo valor da melhor solução do problema da mochila com capacidade W considerando-se só os $n - 1$ primeiros itens.

- Seja $z[k,d]$ o valor ótimo do problema da mochila considerando-se uma capacidade d para a mochila que contém um subconjunto dos k primeiros itens da instância original.
- A fórmula de recorrência para computar $z[k,d]$ para todo valor de d e k é:

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

Mochila - Sobreposição de Subproblemas

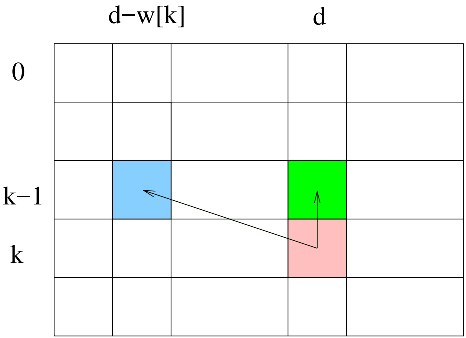
- O problema da mochila possui sobreposição de subproblemas, conforme pode ser observado no exemplo a seguir.
- Considere vetor de tamanhos $w = \{2, 1, 6, 5\}$ e capacidade da mochila $W = 7$. A árvore de recursão seria:



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

- O número total de subproblemas a serem computados é $(n+1)(W+1)$.
- Isso porque tanto o tamanho dos itens quanto a capacidade da mochila são inteiros!
- Podemos então usar programação dinâmica para evitar o recálculo de subproblemas.
- Como o cálculo de $z[k, d]$ depende de $z[k-1, d]$ e $z[k-1, d-w_k]$, preenchemos a tabela linha a linha.

Mochila



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

PD: implementação top-down

```
// A Naive recursive implementation of 0−1 Knapsack problem
#include<stdio.h>
#include <string.h>
// A utility function that returns maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }
// DP matrix declared as a global variable
int z[100][100];

// Returns the maximum value that can be put in a knapsack of capacity W
int knapSack(int W, int w[], int c[], int n)
{
   if (z[n][W]>−1) return z[n][W];    // value already computed
   if (n == 0 || W == 0){             // base Case
     z[n][W]=0;
     return 0;
   }
   // If weight of the nth item is more than Knapsack capacity W, then
   // this item cannot be included in the optimal solution
   if (w[n−1] > W) {
     z[n][W]=knapSack(W, w, c, n−1);
   }
   else  // Return the maximum of two cases: with and without n−th item
     z[n][W]= max(c[n−1] + knapSack(W−w[n−1], w, c, n−1), knapSack(W, w, c, n−1));

   return z[n][W];
} // knapSack
```

PD: implementação top-down (continuação)

```
// Print itens in the optimal solution found in decreasing order of their id's
void WriteSol(int n, int d, int w[]){
  printf(">> Itens in the optimal solution found: ");
  for (int k=n; k>0 && d>0; k--)
    if (z[k][d]!=z[k-1][d]) {
      printf("%d ",k);
      d=d-w[k-1];      // some index adjustments
    }
  printf("\n");
  return;
}

// Driver program to test above function
int main()
{
  int c[] = {10, 7, 25, 24}; // item costs
  int w[] = {2, 1, 6, 5};    // item weights
  int W = 7;                 // knapsack capacity
  int n = sizeof(c)/sizeof(c[0]);

  memset(z,-1,100*100*sizeof(int)); // initialize z

  printf(">> The optimum is: %d\n", knapSack(W, w, c, n));
  WriteSol(n, W, w);

  return 0;
}
```

PD: implementação bottom-up

```
...
// Returns the maximum value that can be put in a knapsack of capacity W
int knapSack(int W, int w[], int c[], int n) {
   int i, j;
   // Build table z[][] in bottom up manner.
   // Attention: the cost (weight) of the i-th item is c[i-1] (w[i-1])
   for (i = 0; i <= n; i++) {
       for (j = 0; j <= W; j++) {
           if (i==0 || j==0) z[i][j] = 0;
           else if (w[i-1] <= j) z[i][j] = max(c[i-1] + z[i-1][j-w[i-1]],  z[i-1][j]);
                else  z[i][j] = z[i-1][j];
       }
   }
   return z[n][W];
}
...
int main() {
    int c[] = {10, 7, 25, 24}; // item costs
    int w[] = {2, 1, 6, 5};    // item weights
    int W = 7;                 // knapsack capacity
    int n = sizeof(c)/sizeof(c[0]);  // number of itens
    memset(z,0,100*100*sizeof(int)); // DIFFERENT FROM the TP code!!!
    printf(">> The optimum is: %d\n", knapSack(W, w, c, n));
    WriteSol(n, W, w);
    return 0;
}
```

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

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

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

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							
3	0							
4	0							

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

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							
4	0							

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

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							

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

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: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

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

Mochila - Complexidade

- Claramente, a complexidade do algoritmo de programação dinâmica para o problema da mochila é $O(nW)$.
- O algoritmo não dá o subconjunto de valor total máximo, apenas o valor máximo.
- Como visto, é fácil recuperar o subconjunto a partir da tabela z preenchida.

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

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

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

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: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

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

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

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

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

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

Mochila - Complexidade

- O algoritmo de recuperação da solução tem complexidade $O(n)$.
- Portanto, a complexidade de tempo e de espaço do algoritmo de programação dinâmica para o problema da mochila é $O(nW)$.
- É possível economizar memória, registrando duas linhas: a que está sendo preenchida e a anterior. Mas isso inviabiliza a recuperação da solução.

Referências

1. S. Halim e F. Halim. Competitive Programming 2, Second Edition Lulu (www.lulu.com), 2011. (IMECC – 005.1 H139c)
2. S. S. Skiena, M. A. Revilla. Programming Challenges: The Programming Contest Training Manual, Springer, 2003.
3. T.H. Cormen, C.E. Leiserson, R.L.Rivest e C. Stein. Introduction to Algorithms. 2nd Edition, McGraw-Hill, 2001. (no. chamada IMECC – 005.133 Ar64j 3.ed.)
4. U. Manber. Introduction to Algorithms: A Creative Approach. Addison-Wesley. 1989. (no. chamada IMECC – 005.133 Ec53t 2.ed.)