PROGRAMAÇÃO DINÂMICA - SUBSET SUM

Prof. Daniel Kikuti

Universidade Estadual de Maringá

2 de outubro de 2014

Elementos de programação dinâmica

Subestrutura ótima

Um subproblema exibe **subestrutura ótima** se uma solução ótima para um problema contém dentro dela soluções ótimas para subproblemas.

Problemas sobrepostos

Quando um algoritmo recursivo reexamina o mesmo problema repetidas vezes, dizemos que o problema de otimização tem **subproblemas sobrepostos**.

Definição do algoritmo

Computar o valor da solução ótima usando tabelas.

Subset sum (soma de subconjunto)

O problema

Dados números naturais $p_1, p_2, \ldots, p_n$ e *sum*, decidir se existe um subconjunto X de $\{1, 2, \ldots, n\}$ tal que:

$$\sum_{i \in X} p_i = sum.$$

O problema admite duas soluções: **sim** e **não**.

Exemplos

- $P = \{30, 80, 30, 20, 40\}$ e $sum = 80$.
- $P = \{3, 1, 4, 12, 5, 7\}$ e $sum = 9$.
- $P = \{1, 2, 3\}$ e $sum = 9$.

Se $n = 0$, o problema tem solução sim se e somente se $sum = 0$.

Subset sum (soma de subconjunto)

O problema

Dados números naturais $p_1, p_2, \ldots, p_n$ e *sum*, decidir se existe um subconjunto X de $\{1, 2, \ldots, n\}$ tal que:

$$\sum_{i \in X} p_i = sum.$$

O problema admite duas soluções: **sim** e **não**.

Exemplos

- $P = \{30, 80, 30, 20, 40\}$ e $sum = 80$. **Sim**. $\{2\}$ ou $\{1, 3, 4\}$.
- $P = \{3, 1, 4, 12, 5, 7\}$ e $sum = 9$. **Sim**. $\{3, 5\}$.
- $P = \{1, 2, 3\}$ e $sum = 9$. **Não**.

Se $n = 0$, o problema tem solução sim se e somente se $sum = 0$.

Estrutura recursiva

Seja $(p_1, \ldots, p_n, sum)$ uma instância do problema. Considere o último elemento n do conjunto de números. Existem duas possibilidades:

- $n \notin X$: X é solução da subinstância $(p_1, \ldots, p_{n-1}, sum)$;
- $n \in X$: $X - \{n\}$ é solução da subinstância $(p_1, \ldots, p_{n-1}, sum - p_n)$.

Definição da recorrência

$$Opt(n, sum) = \begin{cases} Opt(n-1, sum) & n \notin X \\ Opt(n-1, sum - p_n) & n \in X \end{cases}$$

Algoritmo recursivo – Força bruta

```
SUBSET-SUM-REC (P, n, sum)
if n == 0
    if sum == 0 return TRUE
    else return FALSE
else
    s = SUBSET-SUM-REC (P, n-1, sum)
    if s == FALSE e P[n] <= sum
        s = SUBSET-SUM-REC (P, n-1, sum-P[n])
    return s
```

Exercício - Analise o pior caso.

Algoritmo recursivo – Força bruta

```
SUBSET-SUM-REC (P, n, sum)
if n == 0
    if sum == 0 return TRUE
    else return FALSE
else
    s = SUBSET-SUM-REC (P, n-1, sum)
    if s == FALSE e P[n] <= sum
        s = SUBSET-SUM-REC (P, n-1, sum-P[n])
    return s
```

Exercício - Analise o pior caso.

Recorrência $T(n) = 2T(n-1) + \Theta(1)$. O algoritmo examina todos os 2^n subconjuntos de $\{1, \ldots, n\}$.

Sobreposição de problemas

Identificando sobreposições

Exercício. Desenhe a árvore de rerrência para a instância $P = \{4, 2, 1, 3\}$ e $sum = 5$. Observe que a pilha de recursão nunca tem mais que n elementos.

Sobreposição de problemas

Identificando sobreposições

Exercício. Desenhe a árvore de rerrência para a instância $P = \{4, 2, 1, 3\}$ e $sum = 5$. Observe que a pilha de recursão nunca tem mais que n elementos.

Definindo a estrutura memo

- `memo` será uma tabela de dimensões $n \times sum$.
- Inicialmente o valor de cada célula de `memo` é -1.
- `memo[i][k]` é a solução (`TRUE, FALSE`) da instância $(p_1, \ldots, p_i, k)$ do problema.

$$memo[i][k] = \begin{cases} memo[i-1][k] & \text{se } p_i > k \\ memo[i-1][k] \vee memo[i-1][k-p_i] & \text{se } p_i \leq k \end{cases}$$

Algoritmo recursivo – memoizado

```
SUBSET-SUM-MEMO (P, n, sum)
if memo[n][sum] ≠ -1 return memo[n][sum]
else if n == 0
    if sum == 0 memo[n][sum] = TRUE
    else memo[n][sum] = FALSE
else memo[n][sum] = SUBSET-SUM-MEMO (P, n-1, sum)
    if memo[n][sum] == FALSE e P[n] <= sum
        memo[n][sum] = SUBSET-SUM-MEMO (P, n-1, sum-P[n])
    return memo[n][sum]
```

Algoritmo iterativo

```
SUBSET-SUM-ITERATIVO (P, n, sum)
memo[0][0] = TRUE
for k = 1 to n
    memo[0][k] = FALSE
for i = 1 to n
    for k = 0 to sum
        s =  memo[i-1][k];
        if s == FALSE e P[n] <= sum
            s = memo[i-1][k-P[i]]
        memo[i][k] = s
return memo[n][sum]
```

Análise do algoritmo

- O consumo de tempo é proporcional ao tamanho da tabela `memo`.
- O algoritmo consome $\Theta(n * sum)$ unidades de tempo.
- Este algoritmo NÃO é polinomial.
- Se considerarmos a instância do problema $P = 4, 2, 1, 3$ e $sum = 5$, mas agora supondo que seus valores são multiplicados por um valor k, o algoritmo iria consumir k vezes mais tempo para a nova instância.