MAC 5711 - Análise de Algoritmos

Departamento de Ciência da Computação

Primeiro semestre de 2004

Programação Dinâmica

Programação dinâmica é uma outra técnica de projeto de algoritmos que resolve problemas a partir da solução de subproblemas semelhantes.

No método de divisão e conquista, os subproblemas gerados são *independentes*. Mais precisamente, os subproblemas não compartilham subsubproblemas. Por isso, geralmente resolvemos esses subproblemas recursivamente. Quando os subproblemas não são independentes, ou seja, quando eles compartilham subsubproblemas, o método de divisão e conquista deve ser adaptado para que as soluções de subproblemas não sejam recalculadas várias vezes. O resultado dessa adaptação consiste no método de programação dinâmica, que visa garantir que os vários subproblemas sejam resolvidos uma única vez.

A maneira de detectar se programação dinâmica se aplica ou não à resolução de um determinado problema é semelhante ao método de divisão e conquista:

1. determine se a solução de uma instância pode ser calculada a partir da solução de instâncias menores;

2. examine se as instâncias menores de que depende uma dada instância são ou não independentes; isso pode ser feito examinando-se a árvore de recursão do algoritmo recursivo obtido diretamente a partir do passo 1;

3. se a árvore de recursão tem ramos que se repetem, então as instâncias menores não são independentes; se as instâncias são independentes, o método em questão resume-se à divisão e conquista, senão programação dinâmica deve ser mais adequado.

Exemplo 1: Denote por $f(n)$ o número de inteiros cujos dígitos são 1, 2 ou 3 e a soma dos dígitos é n. Escreva um programa que, dado n, imprima $f(n)$.

Vamos ao passo 1 acima. Como podemos determinar $f(n)$ a partir de valores $f(m)$ para $m < n$? De fato, $f(n)$ é dado pela seguinte recorrência:

$$f(n) \quad = \quad \begin{cases} 1 & \text{se } n = 1 \\ 2 & \text{se } n = 2 \\ 4 & \text{se } n = 3 \\ f(n-1) + f(n-2) + f(n-3) & \text{se } n \geq 4. \end{cases}$$

Exercício: Calcule o valor de $f(5)$ pela recorrência acima e mostre os $f(5)$ inteiros cujos dígitos são 1, 2 ou 3 e a soma dos dígitos é 5.

Agora, ao passo 2. Observe que as “instâncias menores”, ou seja, as chamadas $f(n-1)$, $f(n-2)$ e $f(n-3)$ não são independentes, pois produzem “ramos repetidos”: $f(n-1)$ e $f(n-2)$ geram, cada uma, uma chamada de $f(n-3)$. Portanto programação dinâmica se aplica: devemos procurar uma maneira de calcular a solução dessa recorrência evitando recálculos. Uma solução simples neste caso é calcular $f(m)$ iterativamente para m de 1 até n (*bottom-up*), usando para isso três variáveis auxiliares, cada uma guardando um dos três últimos valores calculados da função.

Exercício: Escreva o algoritmo recursivo que calcula $f(n)$ e o algoritmo *bottom-up* descrito acima. Execute as duas versões do algoritmo para alguns valores de n e sinta a diferença no tempo de resposta.

O problema *Counting* (110603) do *Programming Challenges* envolve uma recorrência muito parecida com essa e os mesmos comentários se aplicam.

Exemplo 2: Uma seqüência *bem-formada* de parênteses e sua *profundidade* são definidas indutivamente da seguinte maneira:

- a seqüência vazia de parênteses é bem-formada e tem profundidade 0;
- se A é bem formada, então (A) é bem-formada e tem profundidade um a mais que a profundidade da seqüência A;
- se A e B são bem-formadas, então AB é bem-formada e tem profundidade igual à maior entre as profundidades de A e de B.

Denote por $Q(n,d)$ o número de seqüências bem-formadas com n pares de parênteses e profundidade no máximo d. Escreva um programa que, dados n e d, imprima $Q(n,d)$.

Vamos ao passo 1: como podemos calcular o valor de $Q(n,d)$ a partir de valores de $Q(m,k)$, para m e/ou k menores que n e/ou d? Podemos pensar em uma seqüência bem-formada de n pares de parênteses sempre como um primeiro par de parênteses (recheado por uma seqüência bem-formada de parênteses menor) seguido de uma outra seqüência bem-formada de parênteses (eventualmente vazia). Por exemplo, a seqüência

$$(()(()(())()))()((())())$$

consiste no "primeiro" par de parênteses e seu recheio — $(()(()(())()))$ — e no resto da seqüência — $()((())())$. Tanto o recheio do primeiro par de parênteses quanto o resto da seqüência são seqüências bem-formadas de tamanho menor que a seqüência original: o recheio do primeiro par de parênteses é uma seqüência bem-formada com profundidade no máximo $d-1$, enquanto que o restante da seqüência tem profundidade no máximo d. Quantos pares de parênteses há no primeiro par de parênteses e seu recheio? Não sei... Um número k, que pode ser qualquer coisa entre 1 e n. (Se for n, significa que o restante da seqüência consiste na seqüência vazia.) Como queremos calcular o número total de seqüências bem-formadas com n pares de parênteses e profundidade no máximo d, temos que considerar todos os valores de k, de 1 a n. Para cada valor de k, quantas seqüências há cujo primeiro par de parênteses (e seu recheio) consiste em k pares de parênteses e o restante da seqüência, de $n-k$ pares? Para cada possível recheio do primeiro par de parênteses, podemos escolher uma dentre as várias maneiras diferentes de completar a seqüência. O número de recheios diferentes é $Q(k-1,d-1)$ enquanto que o número de maneiras diferentes de completar a seqüência é $Q(n-k,d)$. Com isso, derivamos a seguinte recorrência para o valor de $Q(n,d)$:

$$Q(n,d) = \begin{cases} 1 & \text{se } n = 0 \\ 0 & \text{se } n > 0 \text{ e } d = 0 \\ \sum_{k=1}^{n} Q(k-1,d-1)Q(n-k,d) & \text{se } n > 0 \text{ e } d > 0. \end{cases}$$

Exercício: Calcule os valores de $Q(5,2)$ e de $Q(5,3)$ pela recorrência acima e mostre as $Q(5,2)$ seqüências bem-formadas com 5 pares de parênteses e profundidade no máximo 2.

Agora vamos ao passo 2, ou seja, perguntemos: a árvore de recursão do algoritmo recursivo para calcular $Q(n,d)$ derivado desta recorrência possui ramos repetidos? A resposta é sim: por exemplo, a chamada $Q(n,d)$ gera uma chamada de $Q(n-2,d-1)$ e uma chamada de $Q(n-1,d)$, que por sua vez também chama $Q(n-2,d-1)$. Portanto programação dinâmica é a escolha indicada neste caso.

E como é o algoritmo obtido por programação dinâmica neste caso? Na verdade, existem vários. O primeiro e talvez mais simples é o que constrói uma *tabela* com os valores de $Q(m,t)$ para todo $0 \leq m \leq n$ e todo $0 \leq t \leq d$, devolvendo ao final o valor da posição (n,d) da tabela. A construção desta tabela segue geralmente o esquema *bottom-up*: preenche-se as entradas da tabela que correspondem à base da recorrência e (como se estivéssemos subindo na árvore de recursão) passa-se a uma entrada (i,j) quando as entradas das quais (i,j) depende já foram preenchidas (solução 1). Uma segunda possibilidade é ainda usar recursão, porém acoplada a um recurso conhecido como *memoização*, que nada mais é do que evitar fazer mais de uma vez a mesma chamada recursiva por meio de uma tabela onde se guarda o resultado de cada chamada recursiva efetuada (solução 2). Algumas vezes o espaço consumido pelo algoritmo de programação dinâmica é um problema. Em muitos casos, é possível evitar o armazenamento da tabela toda. Para tanto, é preciso observar a recorrência e ver qual é a melhor maneira de preenchê-la, de forma que não seja necessário mantê-la toda na memória. Por exemplo, na primeira solução que descrevemos, se a tabela for construída por coluna, ou seja, a coluna 0, depois a coluna 1, depois a coluna 2 e assim por diante (sempre da posição 0 da coluna até a posição n), precisamos apenas da coluna anterior para completarmos a construção da coluna corrente. Ou seja, basta mantermos duas das colunas da matriz e não a matriz toda (solução 3).

Solução 1: Construção da tabela *bottom-up*.

```
int main() {
  int n, d, i, j, k, q[100][100];

  printf("Digite n e d: ");
  scanf("%d %d", &n, &d);
  for (j=0; j<=d; j++)
    q[0][j] = 1;
  for (i=1; i<=n; i++)
    q[i][0] = 0;
  for (j=1; j<=d; j++)
    for (i=1; i<=n; i++) {
      q[i][j] = 0;
      for (k=1; k<=i; k++)
        q[i][j] += q[k-1][j-1]*q[i-k][j];
    }
  printf("Q(%d,%d) = %d\n", n, d, q[n][d]);
  return 0;
}
```

Solução 2: Construção da tabela *top-down* (recursivamente com memoização).

```
int q[100][100];

int bemformada (int n, int d) {
  int k;

  if (q[n][d] == -1)
    if (n == 0)
      q[n][d] = 1;
    else {
```

```
      q[n][d] = 0;
      if (d > 0)
        for (k=1; k<=n; k++)
          q[n][d] += bemformada(k-1,d-1)*bemformada(n-k,d);
    }
  return q[n][d];
}

int main() {
  int n, d, i, j;

  printf("Digite n e d: ");
  scanf("%d %d", &n, &d);
  for (j=0; j<=d; j++)
    for (i=0; i<=n; i++)
      q[i][j] = -1;
  printf("Q(%d,%d) = %d\n", n, d, bemformada(n,d));
  return 0;
}
```

Solução 3: Construção mais econômica em termos de espaço.

```
int main() {
  int n, d, i, j, c, k, q[100][2];

  printf("Digite n e d: ");
  scanf("%d %d", &n, &d);

  /* primeira coluna */
  q[0][0] = 1;
  for (i=1; i<=n; i++)
    q[i][0] = 0;

  /* demais colunas */
  for (j=1; j<=d; j++) {
    /* a coluna j da matriz original será guardada na coluna
       c = j%2 da matriz q, enquanto que a coluna j-1 está em 1-c */
    c = j%2;
    /* primeira entrada da coluna é sempre 1 */
    q[0][c] = 1;
    for (i=1; i<=n; i++) {
      q[i][c] = 0;
      for (k=1; k<=i; k++)
        q[i][c] += q[k-1][1-c]*q[i-k][c];
    }
  }
  printf("Q(%d,%d) = %d\n", n, d, q[n][c]);
  return 0;
}
```

Esses programas e mais o recursivo simples (sem memoização, que consome tempo exponencial) estão disponíveis na página. Divirta-se com eles!