Programação Dinâmica

- **Divisão e conquista:** problema é partido em subproblemas que se resolvem separadamente; solução obtida por combinação das soluções
- **Programação dinâmica:** resolvem-se os problemas de pequena dimensão e guardam-se as soluções; solução de um problema é obtida combinando as de problemas de menor dimensão
- Divisão e conquista é **top-down**
- Programação dinâmica é **bottom-up**
- Abordagem é usual na Investigação Operacional
  - "Programação" é aqui usada com o sentido de "formular restrições ao problema que tornam um método aplicável"
- Quando é aplicável a programação dinâmica: estratégia óptima para resolver um problema continua a ser óptima quando este é subproblema de um problema de maior dimensão

Aplicação directa - Fibonacci

- Problemas expressos recursivamente que podem ser reescritos em formulação iterativa
- **Exemplo:** números de Fibonacci

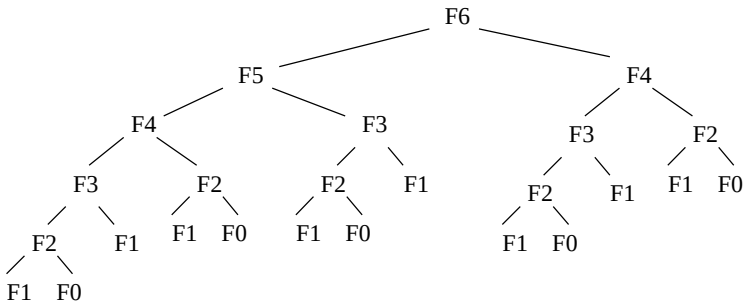
```
/** Números de Fibonacci
 *  versão recursiva
 */ n >= 0

int fib( const unsigned int n )
{
      if( n <= 1 )
            return 1;
      else
            return fib( n-1 ) + fib( n-2 );
}
```

```
/** Números de Fibonacci
 * versão iterativa
 */
int fibonacci(int n )
{
      int last=1, nextToLast=1, answer=1;
      if( n <= 1 )
            return 1;
      for( int i = 2; i<=n; i++ )
      {
            answer = last + nextToLast;
            nextToLast = last;
            last = answer;
      }
      return answer;
}
```

Fibonacci

- Expressão recursiva: algoritmo exponencial
- Expressão iterativa: algoritmo linear

Problema na formulação recursiva: repetição de chamadas iguais



Exemplo: Equação de recorrência

$$C(n) = \frac{2}{n}\sum_{i=0}^{n-1} C(i) + n$$

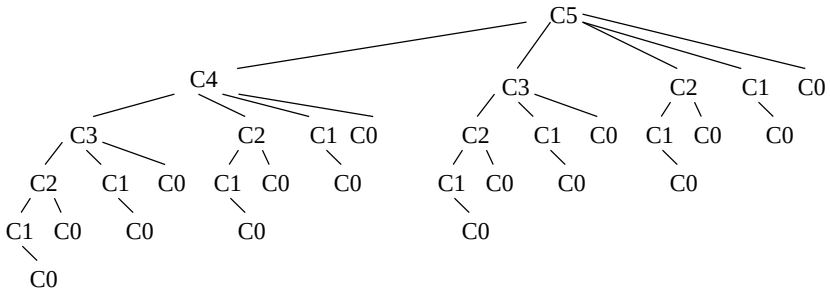
Para resolver numericamente, expressão recursiva é directa

```
double eval( int n )
{
  double sum = 0.0;
  if( n == 0 )
    return 1.0;
  for( int i = 0; i < n; i++ )
    Sum += eval( i );
  return 2.0 * sum / n + n;
}
```

Algoritmo recursivo é exponencial!

Problema: repetição de chamadas

Chamadas Repetidas



Solução iterativa 1

```
double eval(int n )
{
      double [ ] c = new double [n+1];
      c[0] = 1.0;

      for( int i = 1; i <= n; i++ )
      {
            double sum = 0.0;
            for( int j = 0; j < i; j++ )
            sum += c[j];

            c[i] = 2.0 * sum / i + i;
      }
      return c[n];
}
```

Algoritmo iterativo $O(n^2)$

Evita chamadas recursivas guardando tabela de C(n)

Solução iterativa 2

```
double eval(int n )
{
      double sum = 0.0;
      double [ ] a = new double [n+1];
      a[0] = 1.0;

      for( int i = 1; i <= n; i++ )
            a[i] = a[i-1] + 2.0 * a[i-1] / i + i;

       double answer = 2.0 * a[n] / n + n;
      return answer;
}
```

Algoritmo iterativo O(n)

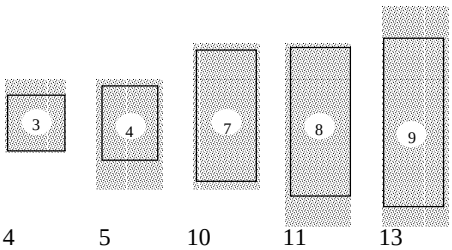
Tabela de A(n) guarda valor dos somatórios; para cada entrada basta acrescentar 1 termo

Problema da mochila

- **Enunciado:**

  O ladrão encontra o cofre cheio de items de vários tamanhos e valores, mas tem apenas uma mochila de capacidade limitada; qual a combinação de items que deve levar para maximizar o valor do roubo?

Exemplo



Valor:	4	5	10	11	13
Nome:	A	B	C	D	E

Mochila: capacidade 17 4 item tipo A: valor 20 1 item tipo D e 1 item tipo E: valor 24

... qual a melhor combinação?

Mochila como Programação Dinâmica

- Muitas situações de interesse comercial
  - melhor forma de carregar um camião ou avião
- Tem variantes: número de items de cada tipo pode ser limitado
- Abordagem programação dinâmica:
  - calcular a melhor combinação para todas as mochilas de tamanho até M
  - cálculo é eficiente se feito pela ordem apropriada

```
for( j = 1; j <= N; j++ )
  {
     for( i=1; i <= M; i++ )
        if ( i >= size[j] )
          if ( cost[i] < cost[i-size[j]] + val[j] )
            {
                cost[i] = cost[i-size[j]] + val[j];
               best[i] = j;
            }
  }
```

Análise do Algoritmo

- `cost[i]` maior valor que se consegue com mochila de capacidade `i`
- `best[i]` último item acrescentado para obter o máximo
- Calcula-se o melhor valor que se pode obter usando só items tipo A, para todos os tamanhos de mochila
- Repete-se usando só A´s e B´s, e assim sucessivamente
- Quando um item `j` é escolhido para a mochila: o melhor valor que se pode obter é `val[j]` (do item) mais `cost[i-size[j]]` (para encher o resto)
- Se o valor assim obtido é superior ao que se consegue sem usar o item `j`, actualiza-se `cost[i]` e `best[i]`; senão mantêm-se.
- Conteúdo da mochila óptima: recuperado através do array `best[i]`; `best[i]` indica o último item da mochila; o restante é o indicado para a mochila de tamanho M-size[best[M]]
- Eficiência: A solução em programação dinâmica gasta tempo proporcional a NM.

EXECUÇÃO

	k	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
j=1	cost[k]	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
	best[k]	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
j=2	cost[k]	4	5	5	8	9	10	12	13	14	16	17	18	20	21	22
	best[k]	A	B	B	A	B	B	A	B	B	A	B	B	A	B	B
j=3	cost[k]	4	5	5	8	10	10	12	14	15	16	18	20	20	22	24
	best[k]	A	B	B	A	C	B	A	C	C	A	C	C	A	C	C
j=4	cost[k]	4	5	5	8	10	11	12	14	15	16	18	20	21	22	24
	best[k]	A	B	B	A	C	D	A	C	C	A	C	C	D	C	C
j=4	cost[k]	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24
	best[k]	A	B	B	A	C	D	E	C	C	E	C	C	D	E	C

Produto de matrizes em cadeia

- **Enunciado:**

  Dada uma sequência de matrizes de dimensões diversas, como fazer o seu produto minimizando o esforço computacional

- Exemplo:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} \begin{bmatrix} c_{11} \\ c_{21} \\ c_{31} \end{bmatrix} \begin{bmatrix} d_{11} & d_{12} \end{bmatrix} \begin{bmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \end{bmatrix}$$

Multiplicando da esquerda para a direita: 84 operações
Multiplicando da direita para a esquerda: 69 operações
Qual a melhor sequência?

Multiplicação de matrizes

- N matrizes a multiplicar $M_1M_2M_3 \ldots M_{N,}$ M_i tem r_i linhas e r_{i+1} colunas
- Multiplicação de matriz pxq por matriz qxr produz matriz pxr, requerendo q produtos para cada entrada - total é pqr operações
- Algoritmo em programação dinâmica:
  - Problemas de dimensão 2: só há 1 maneira de multiplicar, regista-se custo
  - Problemas de dimensão 3:o menor custo de realizar $M_1M_2M_3$ é calculado comparando os custos de multiplicar M_1M_2 por M_3 e de multiplicar M_2M_3 por M_1; o menor é registado.
  - O procedimento repete-se para sequências de tamanho crescente
- Em geral:
  - para $1 \le j \le N-1$ encontra-se o custo mínimo de calcular $M_iM_{i+1} \ldots M_{i+j}$ encontrando, para $1 \le i \le N-j$ e para cada k entre i e i+j os custos de obter $M_iM_{i+1} \ldots M_{k-1}$ e $M_kM_{k+1} \ldots M_{i+j}$ somando o custo de multiplicar estes resultados

Código

```
for( i=1; i <= N; i++ )
    for( j = i+1; j <= N; j++ ) cost[i][j] = INT_MAX;
for( i=1; i <= N; i++ ) cost[i][i] = 0;
for( j=1; j < N; j++ )
    for( i=1; i <=  N-j; i++ )
        for( k= i+1; k <= i+j; k++ )
          {
            t = cost[i][k-1] + cost[k][i+j] +
            r[i]*r[k]*r[i+j+1];
            if( t < cost[i][i+j] )
                { cost[i][i+j] = t; best[i][i+j] = k;
                }
          }
```

Caso geral

- para $1 \le j \le N-1$ encontra-se o custo mínimo de calcular $M_iM_{i+1} ... M_{i+j}$
  - para $1 \le i \le N-j$ e para cada k entre i e i+j calculam-se os custos para obter $M_iM_{i+1} ... M_{k-1}$ e $M_kM_{k+1} ... M_{i+j}$
  - soma-se o custo de multiplicar estes 2 resultados
- Cada grupo é partido em grupos mais pequenos -> custos mínimos para os 2 grupos são vistos numa tabela
- Sendo `cost[l][r]` o mínimo custo para $M_lM_{l+1} ... M_r$,, o custo do 1º grupo em cima é `cost[i][k-1]` e o do segundo é `cost[k][i+j]`.
- Custo da multiplicação final: $M_iM_{i+1} ... M_{k-1}$ é uma matriz $r_i \times r_k$ e $M_kM_{k+1} ... M_{i+j}$ é uma matriz $r_k \times r_{i+j+1}$, o custo de multiplicar as duas é $r_ir_kr_{i+j+k}$.
- Programa obtém `cost[i][i+j]` para $1 \le i \le N-j$, com j de 1 a N-1.
- Chegando a j=N-1, tem-se o custo de calcular $M_1M_2 ... M_N$.
- Recuperar a sequência óptima: array best
  - guarda o rasto das decisões feitas para cada dimensão
  - Permite recuperar a sequência de custo mínimo

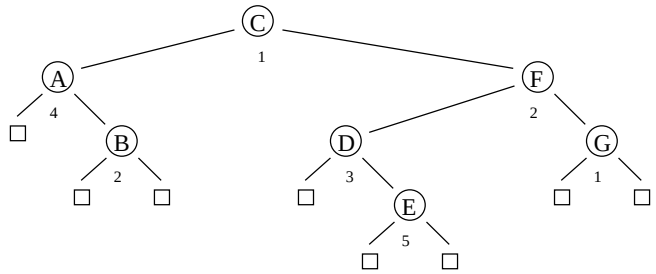
Exemplo - Solução

	B	C	D	E	F
A	24 [A][B]	14 [A][BC]	22 [ABC][D]	26 [ABC][DE]	36 [ABC][DEF]
B		6 [B][C]	10 [BC][D]	14 [BC][DE]	22 [BC] [DEF]
C			6 [C][D]	10 [C][DE]	19 [C] [DEF]
D				4 [D][E]	10 [DE] [F]
E					12 [E] [F]

Árvores de pesquisa binária óptimas

- Em pesquisa, as chaves ocorrem com frequências diversas; exemplos:
  - verificador ortográfico: encontra mais frequentemente as palavras mais comuns
  - compilador de Java: encontra mais frequentemente “if” e “for” que “goto” ou “main”
- Usando uma árvore de pesquisa binária: é vantajoso ter mais perto do topo as chaves mais usadas
- Algoritmo de programação dinâmica pode ser usado para organizar as chaves de forma a minimizar o custo total da pesquisa
- Problema tem semelhança com o dos códigos de Huffman (minimização do tamanho do caminho externo); mas esse não requer a manutenção da ordem das chaves; na árvore de pesquisa binária os nós à esquerda de cada nó têm chaves menores que a deste.
- Problema é semelhante ao da ordem de multiplicação das matrizes

Exemplo



Custo da árvore:
- multiplicar a frequência de cada nó pela sua distância à raiz
- somar para todos os nós

É o **comprimento de caminho interno pesado** da árvore

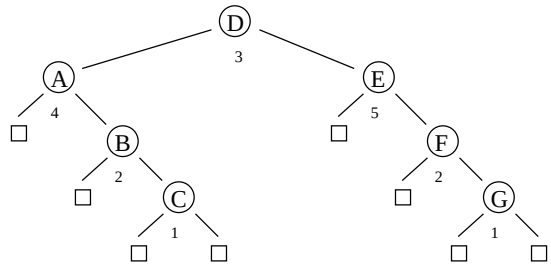
Algoritmo

- Dados:
  - chaves K1 < K2 < ... < Kn
  - frequências respectivas r0, r1, ..., rn
- Pretende-se árvore de pesquisa que minimize a soma, para todas as chaves, dos produtos das frequências pelas distâncias à raiz
- Abordagem em programação dinâmica: calcular, para cada j de 1 a N-1, a melhor maneira de construir subárvore contendo Ki, Ki+1, ..., Ki+j, para $1 \leq i \leq N-j$
- Para cada j, tenta-se cada nó como raiz e usam-se os valores já computados para determinar as melhores escolhas para as subárvores.
- Para cada k entre i e i+j, pretende-se a árvore óptima contendo Ki, Ki+1, ..., Ki+j com Kk na raiz; esta árvore é formada usando a árvore óptima para Ki, Ki+1, ..., Kk-1 como subárvore esquerda e a árvore óptima para Kk+1, Kk+2, ... Ki+j como subárvore direita.

Algoritmo

```
for( i=1; i <= N; i++ )
    for( j = i+1; j <= N+1; j++ ) cost[i][j] = INT_MAX;
for( i=1; i <= N; i++ ) cost[i][i] = f[i];
for( i=1; i <= N+1; i++ ) cost[i][i-1] = 0;
for( j=1; j < N-1; j++ )
    for( i=1; i <=  N-j; i++ )
       {
        for( k= i; k <= i+j; k++ )
          {
            t = cost[i][k-1] + cost[k+1][i+j];
            if( t < cost[i][i+j] )
                { cost[i][i+j] = t; best[i][i+j] = k; }
          }
        for( k= i; k <= i+j; cost[i][i+j] += f[k++] );
       }
```

Árvore binária óptima



Peso da árvore: 41

Eficiência

- O método para determinar uma árvore de pesquisa binária óptima em programação dinâmica gasta tempo $O(N^3)$ e espaço $O(N^2)$
- Examinando o código:
  - O algoritmo trabalha com uma matriz de dimensão N^2 e gasta tempo proporcional a N em cada entrada,
- É possível melhorar:
  - usando o facto de que a posição óptima para a raiz da árvore não pode ser muito distante da posição óptima para uma árvore um pouco menor, no programa dado k não precisa de cobrir todos os valores de i a i+j