1

Técnicas de Concepção de Algoritmos (1ª parte): programação dinâmica

R. Rossetti, A. P. Rocha, L. Ferreira, J. P. Fernandes, F. Ramos, G. Leão
CAL, MIEIC, FEUP

2

Programação dinâmica (*dynamic programming*)

(conclusão…)

Exemplo: ${}^{n}C_k$, versão recursiva

```
long combRec(int n, int k) {
  if (k == 0 || k == n)
    return 1;
  else
    return combRec(n-1, k) + combRec(n-1, k-1);
}
```

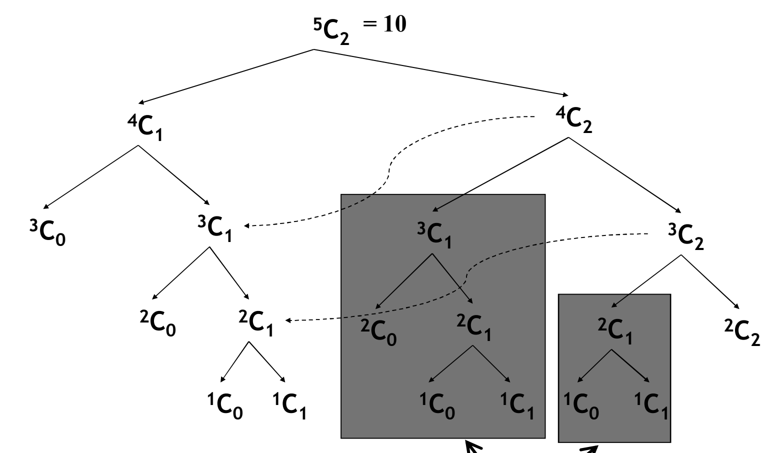
Executa ${}^{n}C_k$-1 vezes (nº de somas a efectuar é nº de parcelas -1)

Executa ${}^{n}C_k$ vezes (nº de 1s / parcelas que é preciso somar)

Executa $2\,{}^{n}C_k$-1 vezes para calcular ${}^{n}C_k$!!

Pode-se melhorar muito, evitando repetição de trabalho (cálculos intermédios ${}^{i}C_j$)

${}^{n}C_k$ - repetição de trabalho



19 nós/chamadas!

8 chamadas duplicadas!

Memorização (*memoization*)

Para economizar tempo, basta aplicar a técnica de memorização (*memoization*), com *array* ou *hash map*.

```
long combMem(int n, int k) {
  // memory to store solutions (initially none)
  static long mem[100][100];  // n <= 99
  // if instance already solved, return from memory
  if (mem[n][k] != 0)
     return mem[n][k];
  // solve recursively
  long sol;
  if (k == 0||k == n) sol = 1;
  else sol = combMem(n-1, k) + combMem(n-1, k-1);
  // memorize and return solution
  mem[n][k] = sol;
  return sol;
}
```


${}^{n}C_{k}$ - Programação dinâmica

Para economizar memória, passa-se a abordagem *bottom-up*.

${}^{n}C_{k}$	k=0	k=1	K=2
n=0	1		
n=1	1	1	
n=2	1	2	1
n=3	1	3	3
n=4	1	4	6
n=5	1	5	**10**

Calculando da esquerda para a direita, basta memorizar uma coluna.

ou

Calculando de cima para baixo, basta memorizar uma linha (diagonal).

7

Implementação

```
long combDynProg(int n, int k) {
  int maxj = n - k;
  long c[1 + maxj];
  for (int j = 0; j <= maxj; j++)
    c[j] = 1;
  for (int i = 1; i <= k; i++)
    for (int j = 1; j <= maxj; j++)
      c[j] += c[j-1];
  return c[maxj];
}
```

n-k+1 vezes

k(n-k) vezes

Tempo: T(n,k) = O(k(n-k))
Espaço: S(n,k) = O(n-k)
(0<k<n, senão O(1))

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

8

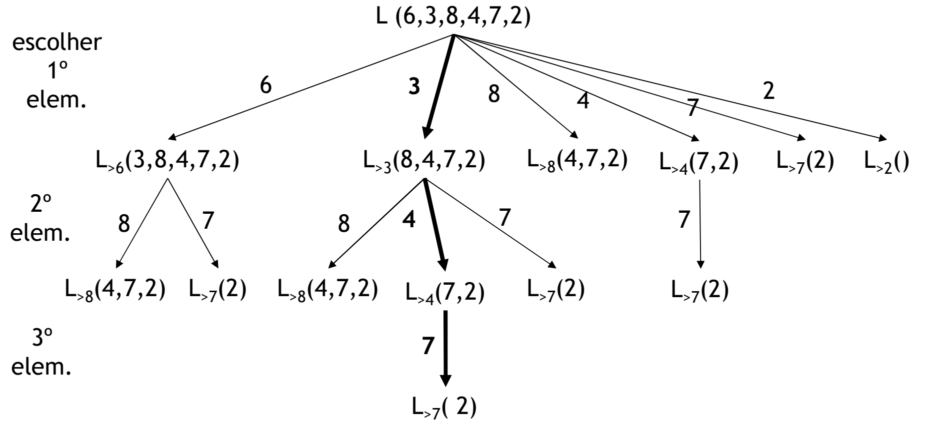
Apêndice:
Exemplos de Derivação de algoritmos

1. Explorar soluções p/ um exemplo
2. Tirar ilações e derivar estratégia
3. Derivar fórmulas de cálculo
4. Derivar algoritmo ou programa

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

9

LIS - Derivação de algoritmo 1º) Explorar soluções p/ um exemplo



Solução ótima: (3, 4, 7)

10

2º) Tirar ilações e derivar estratégia

- Gera subproblemas do tipo $L_{>x}S$, significando “encontrar subsequência crescente mais comprida de *S* com valores maiores do que x” (podendo-se considerar inicialmente *x* = -∞).
- Ocorrem subproblemas repetidos, o que sugere a aplicação de programação dinâmica.
- Subproblemas podem ser identificados pelo índice *i* de *x* na sequência original, ou seja, como L_i.
- Se L_i’s forem resolvidos iterativamente pela ordem L_n, …, L_1, L_0, evita-se repetição de trabalho (programação dinâmica). (No slide anterior, se tivéssemos começado a exploração pelo último elemento, a ordem de iteração seria L_o, L_1,…, L_n.)
- Para cada L_i, em vez de se guardar a solução, basta guardar o tamanho da solução (TL_i) e o índice do 1º elemento da solução (PL_i), e no final reconstrói-se facilmente a solução ótima completa.

11

3º) Derivar fórmulas de cálculo

	i	0	1	2	3	4	5	6=n
Sequência	s_i	$(-\infty)$	6	3	8	4	7	2
Tamanho L_i	TL_i	3	1	2	0	1	0	0
Índ.1ºelem.L_i	PL_i	2	3	4	-	5	-	-

- $TL_i = \max\{1+TL_k \mid i < k \leq n \wedge s_k > s_i\}$ (i=n, …, 0) (max{}=0)
- **PL_i** = valor de k escolhido para o máximo na expressão de TL_i, caso exista, senão "-" (i=n, …, 0)
- Comprimento final: TL_0
- Solução final: s_{PL_0}, $s_{PL_{PL_0}}$, … (parando em "-")
- Neste caso: (s_2, s_4, s_5), isto é, (3, 4, 7)
- Solução "standard" é muito semelhante, mas parte de exploração em sentido inverso (do último para o 1º elemento)!

12

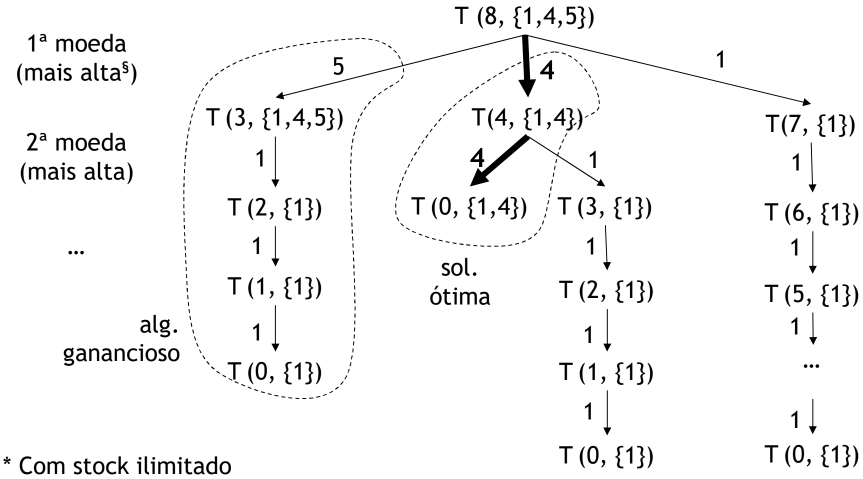
4º) Derivar algoritmo ou programa

```
template <typename T>
void LIS(T s[], int n)
{
   int TL[n + 1] = {0}, PL[n + 1] = {0} /*undef*/;

   for (int i = n; i >= 0; i--)
      for (int k = i + 1; k <= n; k++)
         if (s[k - 1] > s[i - 1] && 1 + TL[k] > TL[i])
         {
            TL[i] = 1 + TL[k];
            PL[i] = k;
         }

   for (int i = PL[0]; i > 0; i = PL[i])
      cout << s[i - 1] << endl;
}
```

$T(n)=O(n^2)$
$S(n)=O(n)$

13

Prob. do troco* – derivação de alg. 1º) Explorar soluções p/ um exemplo



* Com stock ilimitado

§ Para evitar explorar permutações, em cada caminho exploram-se valores não crescentes

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

14

2º) Tirar ilações e derivar estratégia

- No caso geral, algoritmo ganancioso não garante solução ótima.
- Gera subproblemas do tipo *T(k,V)*, significando "encontrar conjunto mínimo de moedas do conjunto *V* de valores unitários que totalizam o montante *k*".
- Ocorrem subproblemas repetidos, o que sugere a aplicação de programação dinâmica.
- Sendo $\{v_1,v_2,…, v_n\}$ o conjunto inicial de valores unitários por ordem crescente, cada subproblema pode ser identificado como *T(i,k)*, significando que se podem usar valores unitários $v_1,…, v_i$.
- Se os *T(i,k)* forem resolvidas iterativamente por ordem crescente de *i* e *k*, evita-se repetição de trabalho (programação dinâmica).
- Para cada *T(i,k),* em vez de se guardar a solução, basta guardar o cardinal (C) da solução e índice (Pi) do maior elemento da solução, e no final reconstrói-se facilmente a solução ótima completa.

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

15

3º) Derivar fórmulas de cálculo

			k=0	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8=m
i=0	v0=-	$C_{0,k}$	0	-	-	-	-	-	-	-	-(ou ∞)
		$P_{0,k}$	-	-	-	-	-	-	-	-	-(ou 0)
i=1	v1=1	$C_{1,k}$	0	1	2	3	4	5	6	7	8
		$P_{1,k}$	-	1	1	1	1	1	1	1	1
i=2	v2=4	$C_{2,k}$	0	1	2	3	1	2	3	4	2
		$P_{2,k}$	-	1	1	1	2	2	2	2	2
i=3	v3=5	$C_{2,k}$	0	1	2	3	1	1	2	3	2
		$P_{2,k}$	-	1	1	1	2	5	5	5	2

- $C_{i,0}$ = **0;** $C_{0,k} = \infty$ **(se** k>0); $P_{0,k} = P_{i,0}$ = indefinido (ou 0)
- $C_{i,k} = C_{i-1,k}$, e $P_{i,k} = P_{i-1,k}$ para i = 1, …, n; k = 1, …, v_i-1
- $C_{i,k} = \min(C_{i-1,k}, 1+C_{i,k-v_i})$ para i = 1, …, n; k = v_i, .., m
- $P_{i,k} = P_{i-1,k}$ ou *i*, conforme se escolhe 1º ou 2º arg. de min
- Cardinal final: $C_{n,m}$ Solução final: $v_{P_{n,m}}$, $v_{P_{n,m-v_{P_{n,m}}}}$, …

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

16

4º) Derivar algoritmo ou programa

```
void troco(int m, int v[], int n)
{
   int C[1 + m] = {0}, P[1 + m] = {0} /*undef*/;
   for (int k = 1; k <= m; k++)
      C[k] = m+1; /*mais alto que qq nº válido*/
   for (int i = 1; i <= n; i++)
      for (int k = v[i]-1; k <= m; k++)
         if (1 + C[k-v[i-1]] < C[k])
         {
              C[k] = 1 + C[k-v[i-1]];
              P[k] = i;
         }
   if (C[m] == m+1)
     cout << "Impossivel" << endl;
   else
     for (int k = m; k > 0; k = k-v[P[k]-1])
        cout << v[P[k]-1] << endl;
}
```

T(n)=O(nm)
S(n)=O(m)

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

17

Em resumo…

- Algoritmos gananciosos (*greedy algorithms*)
  - Contexto: Problemas de optimização (max. ou min.)
  - Objectivo: Atingir a solução óptima, ou uma boa aproximação.
  - Forma: tomar uma decisão óptima localmente, i.e., que maximiza o ganho (ou minimiza o custo) imediato.

- Algoritmos de retrocesso (*backtracking*)
  - Contexto: problemas sem algoritmos eficientes (convergentes) para chegar à solução.
  - Objectivo: Convergir para uma solução.
  - Forma: tentativa-erro. Gerar estados possíveis e verificar todos até encontrar solução, retrocedendo sempre que se chegar a um "beco sem saída".

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

18

Em resumo…

- Divisão e conquista (*divide and conquer*)
  - Contexto: Problemas passíveis de se conseguirem sub-dividir.
  - Objectivo: melhorar eficiencia temporal.
  - Forma: agregação linear da resolução de sub-problemas de dimensão semelhantes até chegar ao caso-base.

- Programação dinâmica (*dynamic programming*)
  - Contexto: Problemas de solução recursiva.
  - Objectivo: Minimizar tempo e espaço.
  - Forma: Induzir uma progressão iterativa de transformações sucessivas de um espaço linear de soluções.

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

19

Referências

- T.H. Cormen, C. E. Leiserson, R. L. Rivest , C. Stein. Introduction to Algorithms, 3rd Edition. MIT Press, 2009
  - Capítulo 15 (Dynamic Programming)
- Mark Allen Weiss. Data Structures & Algorithm Analysis in Java. Addison-Wesley, 1999
- Steven S. Skiena. The Algorithm Design Manual. Springer 1998
- Robert Sedgewick. Algorithms in C++. Addison-Wesley, 1992

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP