1

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

J. Pascoal Faria, Rosaldo Rossetti, Liliana Ferreira
CAL, MIEIC, FEUP
Fevereiro de 2018

2

Programação dinâmica (*dynamic programming*)

Aplicabilidade e abordagem

- Problemas resolúveis recursivamente (solução é uma combinação de soluções de subproblemas similares)
- ... Mas em que a resolução recursiva directa duplicaria trabalho (resolução repetida do mesmo subproblema)
- Abordagem:
  - 1º) Economizar tempo (evitar repetir trabalho), memorizando as soluções parciais dos subproblemas (gastando memória!)
  - 2º) Economizar memória, resolvendo subproblemas por ordem que minimiza nº de soluções parciais a memorizar (*bottom-up*, começando pelos casos base)
- Termo "Programação" vem da Investigação Operacional, no sentido de "formular restrições ao problema que o tornam num método aplicável" e autocontido, de decisão.

Exemplo: ^nC_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 ^nC_k-1 vezes (nº de somas a efectuar é nº de parcelas -1)

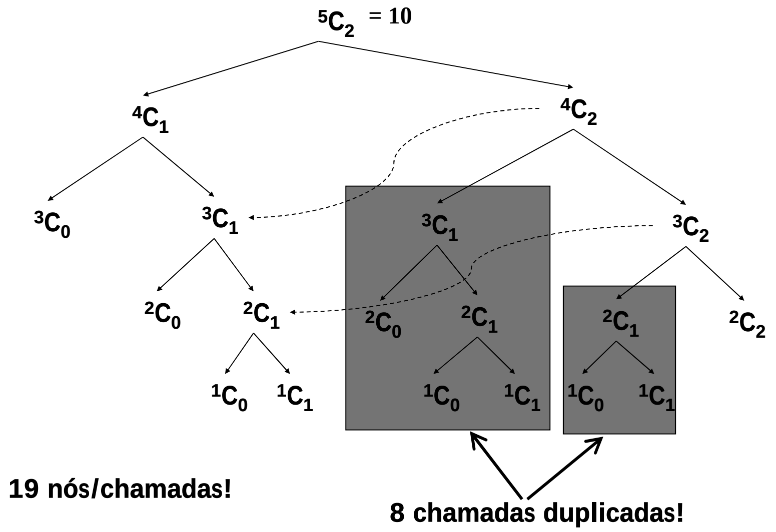
Executa ^nC_k vezes (nº de 1s / parcelas que é preciso somar)

Executa $2\,^nC_k$-1 vezes para calcular ^nC_k !!

Pode-se melhorar muito, evitando repetição de trabalho (cálculos intermédios ^iC_j)

5

^nC_k - repetição de trabalho



Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

6

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;
}
```

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

$^{n}C_{k}$ - 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).

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];
}
```

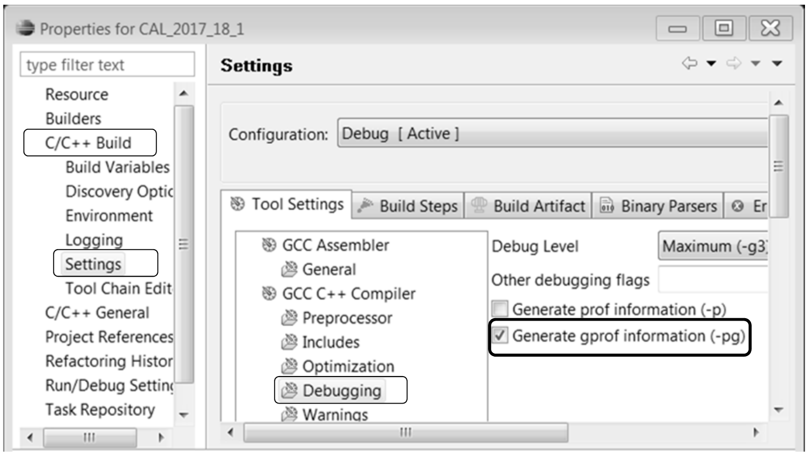
n-k+1 vezes

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))

9

Análise dinâmica de desempenho com *gprof* (*gnu profiler*)



Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

10

```
C:\Users\Jo├o\workspace_cpp\CAL_2017_18_1\Debug>comb 30 15
combRec(30,15)=155117520
combMem(30,15)=155117520
combDynProg(30,15)=155117520

C:\Users\Jo├o\workspace_cpp\CAL_2017_18_1\Debug>gprof -b comb.exe
```

```
self  children    called      name
             310235038            combRec(int, int) [4]
0.77    0.00       1/1          main [3]
0.77    0.00       1+310235038 combRec(int, int) [4]
             310235038            combRec(int, int) [4]
-------------------------------
                 450            combMem(int, int) [5]
0.07    0.00       1/1          main [3]
0.07    0.00       1+450     combMem(int, int) [5]
                 450            combMem(int, int) [5]
-------------------------------
0.00    0.00       1/1          main [3]
0.00    0.00       1         combDynProg(int, int) [77]
-------------------------------
```

Com memorização, o nº de chamadas recursivas baixou de 310235038 para 450 e o tempo baixou de 0.77s para 0.07s.

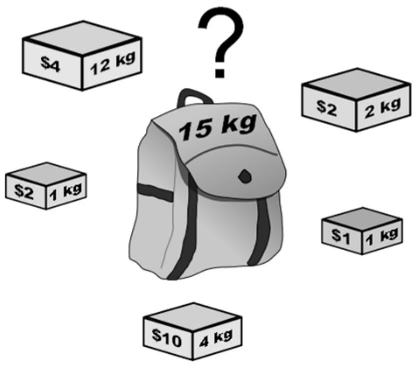
Com programação dinâmica, o tempo baixou para menos de 10 ms.

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

11

Problema da mochila

- Um ladrão encontra o cofre cheio de itens de vários tamanhos e valores, mas tem apenas uma mochila de capacidade limitada; qual a combinação de itens que deve levar para maximizar o valor do roubo?
  - Tamanhos e capacidades inteiros
  - Vamos assumir nº ilimitado de itens de cada tipo



Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

12

Exemplo

Itens

Tamanho	3	4	7	8	9
Valor	4	5	10	11	13
Nome	A	B	C	D	E

Capacidade da mochila: 17

Uma solução óptima:

3	7	7
4	10	10
A	C	C

Outra solução óptima:

8	9
11	13
D	E

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

13

Formalização como problema de programação linear

- Dados
  - m – capacidade da mochila ($m \in \mathbb{N}$)
  - s_1, …, s_n – tamanhos dos itens 1, …, n ($s_i \in \mathbb{N}$)
  - v_1, …, v_n – valores dos itens 1, …, n
- Encontrar valores das **variáveis de decisão**
  - x_1, …, x_n – nº de cópias a usar de cada item ($x_i \in \mathbb{N}$)
- Por forma a maximizar a **função objetivo**: $\sum_{i=1}^{n} v_i x_i$
- Sujeito à **restrição** (inequação): $\sum_{i=1}^{n} s_i x_i \leq m$

Problema de programação linear: problema de otimização em que a função objetivo e as restrições envolvem combinações lineares das variáveis de decisão (no caso geral não resolúvel em tempo polinomial).

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

14

Formulação recursiva

- Necessário para depois aplicar programação dinâmica
- O valor máximo que se consegue colocar numa mochila de capacidade k ($\in \mathbb{N}$), usando itens 1,…, i de tamanho s_1,…, s_i ($\in \mathbb{N}$) e valor v_1,…, v_i pode ser dado pela função:

Usando item i

$$f(i,k) = \begin{cases} 0, & se\ k = 0 \lor i = 0 \\ v_i + f(i, k - s_i), & se\ \ s_i \leq k \land v_i + f(i, k - s_i) > f(i-1, k) \\ f(i-1, k), & noutros\ casos \end{cases}$$

Não usando item i

- O último item na solução ótima é dado pela função:

$$g(i,k) = \begin{cases} 0\ (nenhum), & se\ \ k = 0 \lor i = 0 \\ i, se\ \ s_i \leq k \land v_i + f(i, k - s_i) > f(i-1, k) \\ g(i-1, k), & noutros\ casos \end{cases}$$

- O valor ótimo é $f(n,m)$ c/itens $g(n,m)$, $g(n, m - s_{g(n,m)})$, …

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

Implem. com prog. dinâmica

- Calculando *f* e *g* por ordem de valores de *i* e *k* crescentes, basta memorizar valores para último *i*
- **`f[k]`** e **`g[k]`** na iteração *i* têm valores de *f(i,k)* e *g(i,k)*

```
int f[m+1] = {0}; // iniciado c/ 0's (i=0)
int g[m+1] = {0}; // iniciado c/ 0's (i=0)
for (int i = 1; i <= n; i++ )
  for (int k = s[i]; k <= m; k++)
    if (v[i] + f[k-s[i]] > f[k]) {
       f[k] = v[i] + f[k-s[i]];
       g[k] = i;
    }
// impressão de resultados (valor e itens)
print(f[m]);
for(int k = m; k > 0 && g[k] > 0; k -= s[g[k]])
    print(g[k]);
```

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

Como k é percorrido por ordem crescente f[k-s[i]] já tem o valor da iteração i

*Cálculos para o exemplo dado

	i	s	v	k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	m 17
	0	-	-	f	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
				g	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	3	4	f	0	0	0	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
				g	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	2	4	5	f	0	0	0	4	5	5	8	9	10	12	13	14	16	17	18	20	21	22
				g	0	0	0	1	2	2	1	2	2	1	2	2	1	2	2	1	2	2
	3	7	10	f	0	0	0	4	5	5	8	10	10	12	14	15	16	18	20	20	22	24
				g	0	0	0	1	2	2	1	3	2	1	3	3	1	3	3	1	3	3
	4	8	11	f	0	0	0	4	5	5	8	10	11	12	14	15	16	18	20	21	22	24
				g	0	0	0	1	2	2	1	3	4	1	3	3	1	3	3	4	3	3
n	5	9	13	f	0	0	0	4	5	5	8	10	11	13	14	15	17	18	20	21	23	$\underline{24}$
				g	0	0	0	$\underline{1}$	2	2	1	3	4	5	$\underline{3}$	3	5	3	3	4	5	$\underline{3}$

* Números de Fibonacci

- Formulação recursiva:
  - F(0) = 0
  - F(1) = 1
  - F(n) = F(n-1) + F(n-2), n > 1
- Para calcular F(n), basta memorizar os dois últimos elementos da sequência para calcular o seguinte:

```
int Fib(int n) {
  int a = 1, b = 0 ;  // F(1), F(0)
  for (int i=1; i <= n; i++) {int t = a;   a = b;  b += t; }
  return b;
}
```


Subsequência crescente mais comprida (*LIS - longest increasing subsequence*)

- Exemplo:
  - Sequência s = (9, 5, 2, 8, 7, 3, 1, 6, 4)
  - Subsequência crescente mais comprida (elem's não necessariamente contíguos): (2, 3, 4) ou (2, 3, 6)
- Formulação recursiva 'oficial':
  - $s_1, \ldots, s_n$ - sequência
  - l_i – compr. da maior subseq. crescente de $(s_1, \ldots, s_i)$ terminando em s_i
  - p_i – predecessor de s_i nessa subsequência crescente
  - $\mathbf{l_i = 1 + \max \{ l_k \mid 0 < k < i \wedge s_k < s_i \}}$ (max{} = 0)
  - p_i = valor de k escolhido para o máx. na expr. de l_i
  - Comprimento final: $\max(l_i)$

LIS - Cálculos para o exemplo dado

	i	0	1	2	3	4	5	6	7	8	9
Sequência	si		9	5	2	8	7	3	1	6	4
Tamanho	li		1	1	1	2	2	2	1	3	3
Predecessor	pi		-	-	-	2	2	3	-	6	6

Resposta: (2, 3, 6)

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

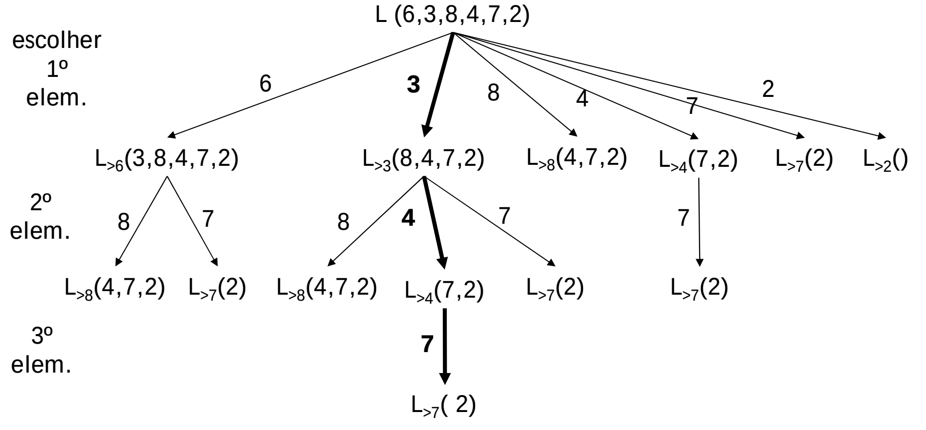
21

APÊNDICE: EXEMPLOS DE DERIVAÇÃO DE ALGORITMOS

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

22

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



Solução ótima: (3, 4, 7)

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

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 $x = -\infty$).
- 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 $L_n, \ldots, L_1, L_0$, evita-se repetição de trabalho (programação dinâmica). (No slide anterior, se tivéssemos começado a exploração pelo último elemento, a ordem de iteração seria $L_0, L_1, \ldots, L_n$.)
- 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.

3º) Derivar fórmulas de cálculo

	i	0	1	2	3	4	5	6=n
Sequência	s_i	(-∞)	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	-	-

- $TL_i = \max\{1+TL_k \mid i < k \le n \wedge s_k > s_i\}$ (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)!

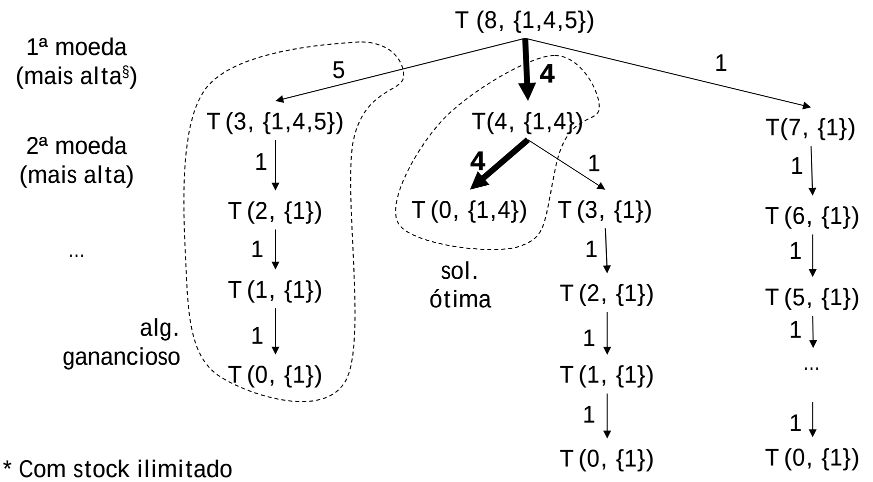
4º) Derivar algoritmo ou programa

```
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;
}
```

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

$S(n)=O(n)$

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

27

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 $\{v_1, v_2, \ldots, v_n\}$ 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 $v_1, \ldots, v_i$.
- 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 (2017-2018)

28

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

- $C_{i,0} = \mathbf{0};$ $C_{0,k} = \infty$ **(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 (2017-2018)

29

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 (2017-2018)