1

Técnicas de Concepção de Algoritmos (1ª parte): Programação Dinâmica

Desenho de Algoritmos

DA, L.EIC

2

Programação dinâmica (*dynamic programming*)

3

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.

Técnicas de Concepção de Algoritmos, DA – L.EIC

4

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)

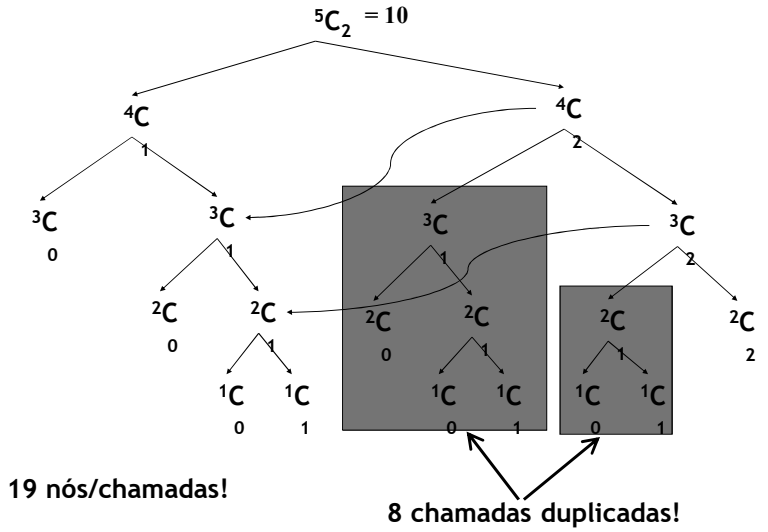
Executa 2^nC_k-1 vezes para calcular ^nC_k !!

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

Técnicas de Concepção de Algoritmos, DA – L.EIC

5

^nC_k - repetição de trabalho



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;
}
```

7

^nC_k - Programação dinâmica

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

^nC_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**

i

j

Para o exemplo $^5C_2$:

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

ou

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

Técnicas de Concepção de Algoritmos, DA - L.EIC

8

Implementação

Guardar apenas uma coluna, e calcular da esq. para dir. :

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

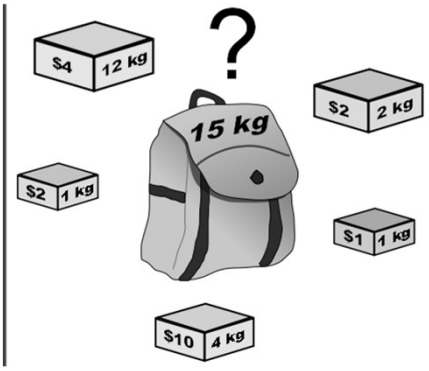
Tempo: T(n,k) = O(k(n-k))
Espaço: S(n,k) = O(n-k)
(0<k<n, senão O(1))

Técnicas de Concepção de Algoritmos, DA - L.EIC

9

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, DA - L.EIC

10

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, DA - L.EIC

11

Formalização como problema de programação linear

- Dados
  - m – capacidade da mochila ($m \in \mathbb{N}$)
  - $s_1, \ldots, s_n$ – tamanhos dos itens 1, …, n ($s_i \in \mathbb{N}$)
  - $v_1, \ldots, v_n$ – valores dos itens 1, …, n
- Encontrar valores das **variáveis de decisão**
  - $x_1, \ldots, x_n$ – 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, DA – L.EIC

12

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 $1,\ldots, i$ de tamanho $s_1,\ldots, s_i$ ($\in \mathbb{N}$) e valor $v_1,\ldots, v_i$ pode ser dado pela função:

Usando item i / Não usando item i

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

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

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

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

Técnicas de Concepção de Algoritmos, DA – L.EIC

13

Estratégia de prog. dinâmica

- Calcular a melhor combinação para todas as mochilas de capacidade k, de 1 até M (capacidade pretendida)
- Começar por considerar que só se pode usar o item 1, depois os itens 1 e 2, etc., e finalmente todos os itens de 1 a N (N = nº de itens)
- Cálculo é eficiente em tempo e espaço se efectuado pela ordem apropriada

14

Evolução dos dados de trabalho

i	s	v
0	-	-
1	3	4
2	4	5
3	7	10
4	8	11
5	9	13

```
Informação sobre os elementos a incluir na mochila

i é o label dado a cada elemento

i = 0 é utilizado para inicializar o procedimento
```

15

Evolução dos dados de trabalho

i	s	v	k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	-	-	f[k]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
			g[k]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	3	4	Iteração:																		
			i = 0;																		
2	4	5	0 <= k <= M;																		
3	7	10																			
4	8	11																			
5	9	13																			

16

Evolução dos dados de trabalho

i	s	v	k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	-	-	f[k]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
			g[k]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	3	4	f[k]	0	0	0	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
			g[k]	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	4	5	Iteração:																		
			i = 1;																		
3	7	10	size[i] <= k <= M;																		
4	8	11	size[i] <= k AND val[i] + cost[k-i] > cost[k]?																		
5	9	13	THEN best[k] = i; cost[k] = val[i] + cost[k-i];																		

17

Evolução dos dados de trabalho

i	s	v	k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	-	-	f[k]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
			g[k]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	3	4	f[k]	0	0	0	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
			g[k]	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	4	5	f[k]	0	0	0	4	5	5	8	9	10	12	13	14	16	17	18	20	21	22
			g[k]	0	0	0	1	2	2	1	2	2	1	2	2	1	2	2	1	2	2
3	7	10	f[k]	0	0	0	4	5	5	8	10	10	12	14	15	16	18	20	20	22	24
			g[k]	0	0	0	1	2	2	1	3	2	1	3	3	1	3	3	1	3	3
4	8	11	f[k]	0	0	0	4	5	5	8	10	11	12	14	15	16	18	20	21	22	24
			g[k]	0	0	0	1	2	2	1	3	4	1	3	3	1	3	3	4	3	3
5	9	13	f[k]	0	0	0	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24
			g[k]	0	0	0	1	2	2	1	3	4	5	3	3	5	3	3	4	5	3

18

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

19

Números de Fibonacci

- **F** = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …}
- Fórmula de recorrência: F(n) = F(n-1) + F(n-2), n > 1
  - F(0) = 0
  - F(1) = 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;
}
```

Técnicas de Concepção de Algoritmos, DA – L.EIC

20

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, …, s_n - sequência
  - l_i – compr. da maior subseq. crescente de (s_1, …, 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)

Técnicas de Concepção de Algoritmos, DA – L.EIC

21

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)

Técnicas de Concepção de Algoritmos, DA - L.EIC

22

Aplicações em Grafos

Técnicas de Concepção de Algoritmos, DA - L.EIC

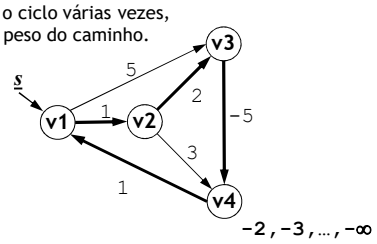
23

Caminho mais curto em grafos dirigidos: Caso de arestas com peso negativo

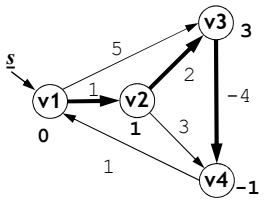
- Neste caso pode ser necessário processar cada vértice mais do que uma vez.
- Se existirem ciclos com peso negativo, o problema não tem solução.
- Não existindo ciclos com peso negativo, o problema é resolúvel em tempo O(|V||E|) pelo **algoritmo de Bellman-Ford** (a seguir).

Sem solução, pois tem um ciclo de peso negativo (-1).

Percorrendo o ciclo várias vezes, diminui-se o peso do caminho.



Com solução, pois não tem ciclos de peso negativo.



Técnicas de Concepção de Algoritmos, DA - L.EIC

24

Algoritmo de Bellman-Ford

- O algoritmo de Bellman-Ford é um exemplo de Programação Dinâmica: começa com um vértice inicial e calcula as distâncias a outros vértices que podem ser alcançados com uma aresta, e continua a encontrar caminhos com duas arestas, e assim sucessivamente.
- Em cada iteração *i*, o algoritmo processa todas as arestas e garante que encontra todos os caminhos mais curtos com até *i* arestas (e possivelmente alguns mais longos) (invariante do ciclo principal).
- Uma vez que o caminho mais comprido, sem ciclos, tem |V|-1 arestas, basta executar no máximo |V|-1 iterações do ciclo principal para assegurar que todos os caminhos mais curtos são encontrados.
- No final é executada mais uma iteração para ver se alguma distância pode ser melhorada; se for o caso, significa que há um caminho mais curto com |V| arestas, o que só pode acontecer se existir pelo menos um ciclo de peso negativo.

- Podem ser efetuadas algumas melhorias ao algoritmo, mas que mantêm a complexidade temporal de O (|V| |E|).

Técnicas de Concepção de Algoritmos, DA - L.EIC

25

Algoritmo de Bellman-Ford

```
BELLMAN-FORD(G, s):  // G=(V,E), s ∈ V
1.   for each v ∈ V do
2.      dist(v) ← ∞
3.      path(v) ← nil
4.   dist(s) ← 0
5.   for i = 1 to |V|-1 do
6.      for each (v, w) ∈ E do
7.         if dist(w) > dist(v) + weight(v,w) then
8.            dist(w) ← dist(v)+ weight(v,w)
9.            path(w) ← v
10.  for each (v, w) ∈ E do
11.     if dist(v) + weight(v,w) < dist(w) then
12.        fail("there are cycles of negative weight")
```

Tempo de execução:

O(|E| |V|)

26

Caminho mais curto entre todos os pares de vértices

- Relevante por exemplo para pré-processamento de um mapa de estradas
- Execução repetida do algoritmo de Dijkstra (ganancioso): $O(|V|\ (|V|+|E|)\ \log|V|)$
  - Bom se o grafo for esparso ($|E| \sim |V|$), como é o caso das redes viárias
- Algoritmo de Floyd-Warshall, programação dinâmica: $\Theta(|V|^3)$
  - Melhor que o anterior se o grafo for denso ($|E| \sim |V|^2$)
  - Mesmo em grafos pouco densos pode ser melhor porque o código é mais simples
  - Baseia-se em matriz de adjacências W[i,j] com pesos (∞ quando não há aresta; 0 quando i = j)
  - Calcula a matriz de distâncias mínimas D[i,j] e a matriz P[i,j] de predecessor no caminho mais curto de *i* para *j*

27

Algoritmo de Floyd-Warshall

- Invariante do ciclo principal: em cada iteração k (de 0 a |V|), D[i,j] tem a distância mínima do vértice i a j, usando apenas vértices intermédios do conjunto {1, …, k}
- Inicialização (k=0):

  $D[i,j]^{(0)} = W[i,j]$ $P[i,j]^{(0)} = nil$
- Recorrência (k=1,…,|V|):

  $D[i,j]^{(k)} = \min(D[i,j]^{(k-1)}, D[i,k]^{(k-1)} + D[k,j]^{(k-1)})$
  - Valor de $P[i,j]^{(k)}$ é atualizado conforme o termo mínimo escolhido
- Para minimizar memória, pode-se atualizar a matriz em cada iteração k, em vez de criar uma nova matriz

28

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".

29

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.

30

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