Notas sobre Programação Dinâmica

José Bernardo Barros

Algoritmos e Complexidade

1 Introdução

Relembremos uma vez mais, a definição da série de Fibonacci, dada pela recorrência

$$F_n = \begin{cases} n & \text{se } n < 2 \\ F_{n-1} + F_{n-2} & \text{se } n \geq 2 \end{cases}$$

Uma definição naïve da função que calcula o n-ésimo elemento desta série é dada por

```
int fib1 (int n) {
  int f1, f2;

  if (n<2) return n;
  f1 = fib1 (n-1);
  f2 = fib1 (n-2);
  return (f1+f2);
}
```

Uma análise da complexidade assimptótica desta função revela-nos que o seu tempo de execução é exponencial no valor do seu argumento. Esta pouca eficiência deve-se à (excessiva) repetição de cálculos.
Repare-se que, por exemplo, ao calcularmos `fib 10`, a função vai calcular `fib 1` 55 vezes!
Uma forma de obtermos uma definição mais eficiente desta função consiste em evitar a duplicação de cálculos, guardando os cáculos já efectuados (para calcularmos `fib n` vamos precisar de calcular todos os valores da função `fib` para argumentos menores do que `n`).
Podemos por isso apresentar uma definição alternativa.

```
int fibAux (int n, int fibs[]) {
  int f1, f2;
  if (fibs[n] == -1)
    if (n<2) fibs [n] = n;
    else {
      f1 = fibAux(n-1,fibs);
      f2 = fibAux(n-2,fibs);
      fibs[n] = f1 + f2;
    }
  return (fibs[n]);
}
```

Esta função auxiliar `fibAux` retorna o valor de F_n; caso ele já tenha sido calculado (e por isso encontra-se no array `fibs`, apenas retorna esse valor. Caso contrário, calcula o valor e coloca-o no array. Dessa forma garantimos que nenhum valor é realmente recalculado.

```
int fib2 (int n) {
  int fibs [n+1], int i;

  for (i=0;i<n;i++) fibs [i] = -1;
  return (fibAux (n, fibs));
}
```

A função que calcula F_n usando esta função auxilar apenas precisa de inicializar o array `fibs`. A aparente proximidade das definições das funções `fib1` e `fibAux` pode nos fazer pensar que não houve ganho de eficiêcia. No entanto, e para mostrarmos que esta definição é de facto muito mais eficiente devemos reparar que as chamadas recursivas só são feitas se esta for a primeira vez que estamos a calcular o valor em causa (é esse o propósito de começarmos por inicializar todos os valores do array com `-1`).

Esta técnica de optimização que consiste em evitar cálculos repetidos pelo armazenamento dos resultados desses cálculos é conhecida por *memoization* e implementada por muitas linguagens de programação.

No entanto podemos ainda aplicar outras técnicas para aumentar a eficiência da definição acima. Para isso podemos notar que apesar do cálculo de F_n necessitar dos valores de F_k para $k < n$, existe uma ordem de cálculo dos vários F_k que nos garante que todos os que são necessários serem calculados já o foram. Daí que seja possível reescrever a definição acima sem a preocupação de verificar se um dado valor já está ou não calculado.

```
int fib3 (int n){
  int fibs[n+1], i;

  fibs[0] = 0; fibs[1] = 1;
  i = 1;
  while (i<n) {
       i = i+1;
       fibs[i] = fibs[i-1]+fibs[i-2];
  }
  return fibs[n];
}
```

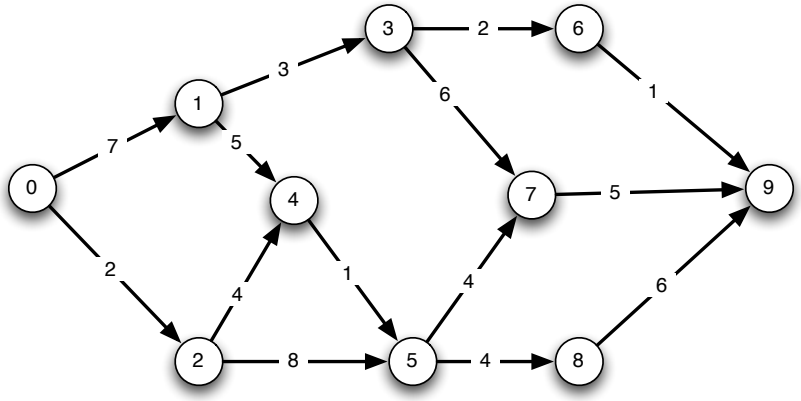
Este tipo de optimização que consiste em determinar uma ordem de cálculo dos resultados parciais de forma a que estes resultados estejam calculados quando necessários é normalmente conhecido por **programação dinâmica**.
No caso que estamos a analisar, podemos constatar que, para calcular F_n não precisamos de guardar todos os F_k; precisamos apenas de guardar os dois últimos elementos da série. Daí que possamos reescrever a definição acima de forma a evitar usar um array:

```
int fib4 (int n){
  int f,f1,f2, i;
```

```
  if (n<2) return n;
  f1=1; f2=0;
  i = 1;
  while (i<n) {
    i = i+1;
    f = f1 + f2;
    f2=f1; f1=f;
  }
  return f;
}
```

2 Caminho mais longo

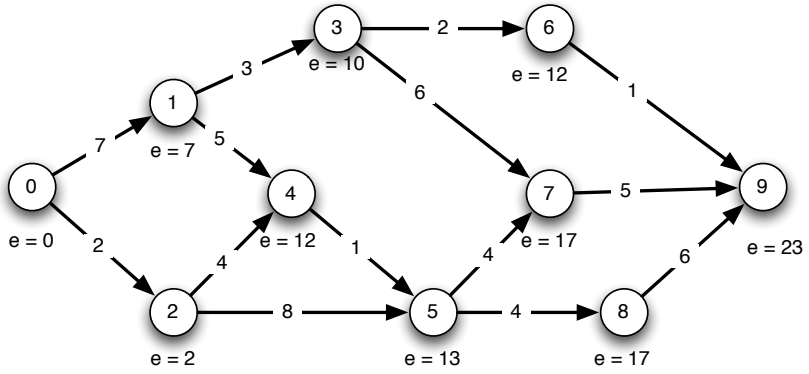
Um outro exemplo característico desta técnica de optimização é o cálculo do caminho mais longo num grafo pesado. No caso geral este problema é um problema difícil (*NP-hard*). No entanto tem uma solução linear se o grafo fôr orientado e acíclico (*Directed Acyclic Graph*). Vejamos um exemplo. Seja G o seguinte grafo



A restrição de o grafo ser acíclico permite-nos obter uma ordenação topológica dos vértices do grafo.
No grafo acima, uma possível ordenação topológica dos vértices é:

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9$$

Esta ordenação pode ser usada para *escalonar* a ordem pela qual vamos analisar o grafo.
Assim, e seguindo esta sequência, vamos calcular o caminho mais longo que contem apenas os vértices já visitados.
Para isso, e sempre que visitamos um nodo v, testamos se, para cada um dos seus sucessores x (e que por isso ainda não foram visitados) o caminho correspondente ao maior caminho até v seguido de x é um caminho maior do que o descoberto até à altura.
Precisamos então de guardar, para cada nodo x o custo do maior caminho (até à altura) terminado em x; vamos denotar esse valor por e_x.
No final deste processo os custos calculados para cada vértice são:



Assumindo a existência de uma função que calcula uma ordenação topológica de um grafo, esta função pode ser definida como:

```
int longestPath (Grafo g){
  int OT[N], e[N], i, r;

  ordTop (g,OT);
  for (i=0;i<N;e[i++]=0);
  for (i=0;i<N;i++){
    v = OT[i];
    for (x in sucessores(v))
      if (e[v]+g(v,x) > e[x])
        e[x]=e[v]+g(v,x);
  }
  r=0;
  for (i=0;i<N;i++)
    if (e[i] > r) r = e[i];
  return r;
}
```

Esta função calcula o custo do caminho mais longo num grafo (orientado e acíclico). Para calcularmos o caminho podemos percorrer os vértices do grafo pela ordem inversa de forma a calcular quais desses vértices pertencem ao caminho.

A função apresentada acima calcula, para cada vértice o custo do caminho mais longo de um dos vértices iniciais (sem antecessores) a esse vértice.
Uma pequena modificação pode ser usada para, nas mesmas condições (i.e., num grafo orientado e acíclico) determinar o peso do caminho mais curto entre dois vértices. Mais uma vez podemos usar a ordenação topológica do grafo para implementar um algoritmo baseado em programação dinâmica.

3 Fecho Transitivo

Na sua versão mais simples (grafo não pesado) este problema consiste em, dado um grafo G, calcular um grafo G^+, no qual existe uma aresta do vértice a para o vértice b sse existe em G um caminho entre esses vértices.

Uma solução para este problema baseada em programação dinâmica (e conhecida por algoritmo de Warshal) consiste em, para um grafo G com vértices $\{0..,N-1\}$, definir G_i^+ como o grafo no qual existe uma aresta do vértice a para o vértice b sse existe em Gum caminho entre esses vértices cujos vértices intermédios são $< i$.
De acordo com esta definição, o grafo G^+ não é nada mais do que G_N^+.
Note-se que de acordo com esta definição

- existe uma aresta do vértice a para o vértice b em G_0^+ sse existe uma aresta do vértice a para o vértice b em G, i.e., $G_0^+ = G$
- G_i^+ pode ser calculado apenas por consulta a G_{i-1}^+, pelo que não precisamos de guardar os vários G_i^+.

```
void tclosure (Grafo g, int GP[N][N]){
  int i, a, b;

  for (a=0; a<N; a++)
    for (b=0; b<N; b++)
      GP[a][b] = 0;
  for (a=0; a<N; a++)
    for (b in sucessores (a))
      GP[a][b] = 1;

  for (i=0; i<N; i++)
    for (a=0;a<N;a++)
      if (GP[a][i])
        for (b=0; b<N; b++)
          if (GP [i][b])
            GP[a][b] = 1;
}
```

Uma variante deste algoritmo (Floyd-Warshal), calcula, para um grafo pesado G, os pesos dos caminhos mais curtos entre cada par de vértices.
A estratégia usada é similar: definimos G_i^+ como o grafo pesado em que existe uma aresta de a para b com peso w sse o peso do caminho mais curto entre a e b passando apenas por vértices intermédios $< i$ é w.
Mais uma vez podemos constatar que:

- existe uma aresta do vértice a para o vértice b com peso w em G_0^+ sse existe uma aresta do vértice a para o vértice b com peso w em G, i.e., $G_0^+ = G$
- G_i^+ pode ser calculado apenas por consulta a G_{i-1}^+, pelo que não precisamos de guardar os vários G_i^+.

```
void floydWarshal (Grafo g, int GP[N][N]){
  int i, a, b, w;

  for (a=0; a<N; a++)
    for (b=0; b<N; b++)
```

```
      GP[a][b] = 0;
  for (a=0; a<N; a++)
    for (b in sucessores (a))
      GP[a][b] = g(a,b); // peso da aresta a->b

  for (i=0; i<N; i++)
    for (a=0;a<N;a++)
      if (GP[a][i])
        for (b=0; b<N; b++)
          if (GP [i][b]){
            w = GP[a][i] + GP[i][j];
            if ((GP[a][b] == 0) ||
                (GP [a][b] > w))
              GP[a][b] = w;
          }
}
```

4 Vendedor ambulante

O problema do *vendedor ambulante* é um problema conhecido por ter em geral uma solução difícil. Considere-se que um vendedor ambulante tem que decidir quais os produtos que levará na sua próxima viagem. Infelizmente tem um limite de peso que pode transportar. Tendo isso em atenção terá que escolher a combinação de produtos que, sem exceder esse peso, lhe permitirá maximizar os seus lucros.
O input deste problema consiste na descrição dos produtos disponíveis, incluindo o peso W_i e lucro P_i associado a cada produto i, bem como do limite de peso a transportar w.
Uma versão simplificada deste problema tem como output apenas o valor do lucro máximo que pode ser atingido. Uma versão mais completa passa por calcular a lista dos produtos a incluir nessa viagem.
Uma abordagem baseada em programação dinâmica pode ser desenvolvida a partir da definição de L_i como o lucro máximo que pode ser obtido transportando objectos cujo peso é $\leq i$. Note-se que, de acordo com esta definição L_i pode ser calculado a partir dos L_j com $j < i$. Além disso, o resultado pretendido não é mais do que L_w.

```
int knapsack (int W[], int P[], int N, int w) {
  int L[w+1], i, p, m;
  L[0] = 0;
  for (i=1; i<=w; i++){
    L[i] = L[i-1];
    for (p=0; p<N; p++){
      if ((W[p] <= i) && (L[i] < P[p] + L[W[p]]))
        L[i] = P[p] + L[W[p]];
    }
  }
  return L[w];
}
```

5 Distância de Edição

O problema do cálculo da distância mínima de edição entre duas *strings* (ou mais genéricamente entre duas sequências) consiste em, dadas duas sequências, determinar o menor número de operações elementares que são necessárias para transformar uma das sequências na outra.
As operações elementares em causa são:

- inserção de um elemento numa dada posição
- remoção de um elemento de uma dada posição
- substituição de um elemento por outro

Por exemplo, as *strings* ... e ... estão a uma distância de ...:
Uma solução para este problema, usando programação dinâmica, consiste em, dadas as duas sequências $\{S_i\}_{0 \leq i < N}$ e $\{T_i\}_{0 \leq i < M}$, definir $D_{p,q}$ como a distância mínima de edição entre os prefixos $\{S_i\}_{0 \leq i < p}$ e $\{T_i\}_{0 \leq i < q}$.
Note-se que o resultado que estamos a tentar descobrir, i.e., a distância de edição entre as duas sequências, nada mais é do que $D_{N,M}$.
Vejamos então como calcular $D_{p,q}$, assumindo que conhecemos todos os outros $D_{i,j}$ com $i \leq p$ e $j \leq q$. Em particular, vamos admitir que conhecemos $D_{i-1,j}$, $D_{i,j-1}$ e $D_{i-1,j-1}$. Comparando os elementos S_{i-1} e T_{j-1} temos então as seguintes alternativas para o valor de $D_{i,j}$:

1. Se S_{i-1} e T_{j-1} forem iguais, podemos calcular $D_{i,j}$ apenas como $D_{i-1,j-1}$
2. Se S_{i-1} e T_{j-1} forem diferentes, temos 3 alternativas para o cálculo de $D_{i,j}$:
   (a) inserir S_{i-1} em T: resultando numa distância de edição de $1 + D_{i-1,j}$;
   (b) inserir T_{i-1} em S, resultando numa distância de edição de $1 + D_{i,j-1}$;
   (c) substituir S_{i-1} por T_{j-1} (ou vice-versa), resultando numa distância de edição de $1 + D_{i-1,j-1}$.

Como pretendemos calcular a distância mínima de edição teremos que calcular qual destas alternativas resulta num menor valor.
Note-se finalmente que as distâncias mínimas de edição entre uma sequência vazia e uma qualquer sequência é o comprimento dessa sequência.
A solução que a seguir se apresenta resolve o problema para arrays de inteiros. É fácil adaptá-la para outro tipo de arrays. No caso das strings, em que o tamanho da string não é conhecido antes de a percorrer, é necessário começar por calcular esses tamanhos.

```
int edit_distance (int S[], int N, int T[], int M){
  int D[N+1][M+1], i, j;
  for (i=0;i<=N;i++) D[i][0] = i;
  for (j=0;j<=M;j++) D[0][j] = j;
  for (i=1;i<=N;i++)
    for (j=1;j<=M;j++){
      if (S[i-1] == S[j-1])
        D[i][j] = D[i-1][j-1];
      else D[i][j] = 1 + D[i-1][j-1];
      if (D[i][j] > 1+D[i][j-1])
```

```
        D[i][j] = 1+D[i][j-1];
      if (D[i][j] > 1+D[i-1][j])
        D[i][j] = 1+D[i-1][j];
    }
  return D[N][M];
}
```

6 Max subseq sum

Dada uma sequência de números, o cálculo da sub-sequência contígua que tem a maior soma também pode ser resolvido usando programação dinâmica.
Para isso, e dada uma sequência S com N elementos, podemos definir a sequência M também com N elementos em que M_i tem o valor da maior soma da subsequência de S que termina em S_i.
É fácil verificar que a sequência M pode facilmente ser definida pela seguinte recorrência ($M_0 = S_0$):

$$M_i = \begin{cases} M_{i-1} + S_i & \text{se} \quad M_{i-1} > 0 \\ S_i & \text{se} \quad M_{i-1} \leq 0 \end{cases}$$

Esta definição é suficiente para justificar a seguinte definição em C:

```
int max_subseq_sum (int S[], int N){
  int M[N], i, r;
  M[0] = S[0];
  for (i=1;i<N;i++)
    if (M[i-1]>0) M[i] = M[i+1] + S[i];
    else M[i] = S[i];
  r = M[0];
  for (i=1;i<N;i++)
    if (M[i]>r) r = M[i];
  return r;
}
```

Tal como fizemos no exemplo da sequência de Fibonacci, também aqui podemos evitar o uso do array auxiliar. Isto porque, de facto, cada elemento desse array depende apenas do elemento anterior. Além disso, o cálculo do maior elemento do array pode ir sendo feito à medida que esses valores são calculados.

```
int max_subseq_sum2 (int S[], int N){
  int M, i, r;
  r = M = S[0];
  for (i=1;i<N;i++){
    if (M>0)
       M = M + S[i];
    else M = S[i];
    if (M>r) r = M;
  }
  return r;
}
```

7 Subset sum

O último exemplo que vamos analisar é o de, dado um conjunto (sequência) de números determinar se a soma de alguns desses números prefaz um determinado valor.
Uma possível definição desta função baseia-se na seguinte recorrência:

- Se o conjunto for vazio, a única forma de a função retornar verdadeiro é se pretendermos que a soma prefaça 0.
- Para um conjunto não vazio, temos duas alternativas para obter a soma desejada:
  - não incluindo o 1º elemento, i.e., obtendo a soma desejada com todos os outros elementos, ou
  - incluindo o 1º elemento e tentando prefazer com os restantes elementos a restante soma.

Esta recorrência, sob a forma de uma definição recursiva em C vem:

```
int subset_sum (int v[], int N, int x){
  if (x == 0) return 1;
  if (N == 0) return 0;
  return (subset_sum (v+1,N-1,x) ||
          subset_sum (v+1,N-1,(x-v[0])));
}
```

Esta definição é exponencial no tamanho `N` do array argumento.
Em alguns casos é possível obter uma solução mais eficiente baseada em programação dinâmica.
Vamos para isso começar por definir um predicado S como

> $S(i, j)$ é verdadeiro sse é possível obter a soma i usando os primeiros j elementos da sequência original.

Tendo em conta que para obtermos a soma i podemos ou não usar o elemento na posição j, este predicado pode ser definido indutivamente (assumindo que os índices do array se iniciam em 0):

$$S(i, j) = S(i, j-1) || S(i - \texttt{v[j-1]}, \texttt{j-1})$$

Se tivermos a garantia que tanto a soma objectivo como os elementos do conjunto (array) são todos não negativos, a definição indutiva acima pode ser usada para escalonar os cálculos necessários.

```
int subset_sum_pos (int v[], int N, int x){
  int S[x+1][N+1], i, j;

  for (j=0;j<=N;j++) {
    S[j][0] = 0;
    S[0][j] = 1;
  }
  for (j=1; j<=N; j++)
    for (i=0;i<=x;i++)
      S[i][j] = S[i][j-1] ||
```

```
                    ((v[j-1]<=i) &&
                     (S[i-v[j-1]][j-1]));
  return (s[x][N]);
}
```

Exercícios

1. Tal como referido na secção 2, apresente uma definição da função baseada em programação dinâmica que calcula o caminho mais curto entre dois vértices num grafo orientado e acíclico.

2. Uma variante do algoritmo de Floyd-Warshall apresentado na secção 3 guarda numa matriz `I` os vértices intermédios dos caminhos mais curtos.

   Apresente uma definição dessa variante bem como uma definição da função que, dada uma dessas matrizes e dois vértices, calcula a sequência de vértices que constituem o caminho mais curto entre esses vértices.

3. Uma variante do problema do vendedor ambulante apresentado na secção 4 consiste em não permitir repetições, i.e., que cada produto só pode ser incluido uma vez na lista final. Apresente uma definição para resolver este problema baseada em programação dinâmica.

   Sugestão: Defina, $L_{i,j}$ como o lucro máximo que pode ser obtido transportando objectos $\leq j$ e cujo peso é $\leq i$.

4. A função `subset_sum_pos` apresentada na secção 7 pode ser melhorada se usarmos *momoization*. Para isso, e em vez de usar a matriz `M`, podemos usar uma tabela de hash para implementar conjuntos de pares (predicado binário).

5. Apresente uma definição alternativa da função `max_subseq_sum` apresentada na secção 6 de forma a, em vez de calcular o valor da maior soma de elementos consecutivos, escreva no ecran esses valores.

   Sugestão: Defina, para cada posição i do array, P_i como sendo a posição onde se inicia a sequência de maior soma que termina e inclui o elemento na posição i.

   Apresente uma definição alternativa baseada nesta sugestão.

6. Defina uma função que, dado um array `v` de `N` inteiros, calcula o comprimento da maior subsequência crescente de elementos de `v`. Note que os elementos da sequência não têm que ser necessáriamente contíguos.

   Sugestão: Defina, para cada posição i do array, S_i como sendo o tamanho da maior subsequência do array que termina e inclui `v[i]`.