Departamento de Ciência de Computadores FCUP

Desenho e Análise de Algoritmos (CC211) 2013/14

1º Teste Escrito (13.11.2013) *duração: 2h30* **Cotação: 1+4+3, 4×1.5, 4+2**

UMA RESOLUÇÃO (a adaptar para as outras versões do enunciado)

1. Considere o problema de ordenar por **ordem crescente** um vetor com n inteiros, dados nas posições $v[0], v[1], \ldots, v[n-1]$, com $n \geq 2$, e o seguinte algoritmo para o resolver.

```
1. Para k ← 1 até n − 1 fazer
2.     x ← v[k];
3.     j ← k − 1;
4.     Enquanto (j ≥ 0 ∧ x < v[j]) fazer
5.         v[j + 1] ← v[j];
6.         j ← j − 1;
7.     v[j + 1] ← x;
```

a) Para a instância $v = [4, -5, 30, -1, 2], n = 5$, qual é o estado de v, de x e de j após a execução do **bloco 2.-6.**, para $k = 3$?

Resposta:

$v = [-5, 4, 4, 30, 2]$, $x = -1$ *e* $j = 0$.

b) Analise a complexidade temporal do algoritmo. Apresente as conclusões usando as notações Θ, O, e Ω (deve dar uma resposta elucidativa e útil). Justifique, indicando: um modelo de tempos para instruções básicas; expressões que definem o tempo total que o algoritmo demora, no melhor e no pior caso (e descreva casos desses); e provas de que o tempo pertence às classes que referir.

Resposta:

O algoritmo tem complexidade temporal $O(n^2)$, sendo, no melhor caso, de ordem $\Theta(n)$ e, no pior, de ordem $\Theta(n^2)$. O melhor caso ocorre quando o vetor está ordenado por ordem crescente (as instruções 5.–6. não são executadas). O pior caso ocorre quando o vetor está ordenado por ordem estritamente decrescente (as instruções 5.–6. são executadas o número máximo de vezes, k vezes para cada k, com $1 \leq k \leq n-1$). Para ver que assim é, designemos por $T(n)$ o tempo total numa instância de tamanho n e por $c_1, \ldots, c_9$ os tempos de execução das instruções básicas assim definidas:

- c_1 **:** *atribuição de constante a variável simples ($k \leftarrow 1$);*
- c_2 **:** *teste da condição $k \leq n-1$ de paragem do ciclo "para" e transferência de controlo;*
- c_3 **:** *atribuição $j \leftarrow j+1$, ou $j \leftarrow k-1$, ou $j \leftarrow j-1$;*
- c_4 **:** *atribuição $x \leftarrow v[k]$;*
- c_5 **:** *teste da 1ª condição de paragem do ciclo "Enquanto" ($j \geq 0$) e transferência de controlo;*
- c_6 **:** *teste da 2ª condição de paragem do ciclo "Enquanto" ($x < v[j]$) e transferência de controlo;*
- c_7 **:** *atribuição $v[j+1] \leftarrow v[j]$;*
- c_8 **:** *atribuição $v[j+1] \leftarrow x$;*
- c_9 **:** *transferência de controlo no final de cada iteração (no ciclo "Para" e "Enquanto").*

No melhor caso, $T(n) = (c_1 + c_2 n + (c_3 + c_9)(n-1)) + (c_4 + c_3 + c_8)(n-1) + (c_5 + c_6)(n-1)$. Logo, para $\alpha = \min\{c_1, \ldots, c_9\}$ e $\beta = \max\{c_1, \ldots, c_9\}$, tem-se

$$\alpha n < \alpha + \alpha n + 7\alpha(n-1) \leq T(n) \leq \beta + \beta n + 7\beta(n-1) < 9\beta n$$

para todo $n \geq 1$. Para concluir que $T(n) \in \Theta(n)$, isto é, que existem constantes $C, C' \in \mathbb{R}^+$ e $n_0 \in \mathbb{Z}^+$ tais que $Cn \leq T(n) \leq C'n$, para todo $n \geq n_0$, basta tomar, $C = \alpha$ e $C' = 9\beta$ e $n_0 = 1$.

No pior caso,

$$T(n) = (c_1 + c_2 n + (c_3 + c_9)(n-1)) + (c_4 + c_3 + c_8)(n-1) + \sum_{k=1}^{n-1}((c_3 + c_7 + c_5 + c_6 + c_9)k + c_5).$$

(CONTINUA, v.p.f.)

Logo,

$$\alpha + \alpha n + 5\alpha(n-1) + \sum_{k=1}^{n-1}(5\alpha k + \alpha) \leq T(n) \leq \beta + \beta n + 5\beta(n-1) + \sum_{k=1}^{n-1}(5\beta k + \beta)$$

ou seja,

$$\alpha + \alpha n + 6\alpha(n-1) + 5\alpha \sum_{k=1}^{n-1} k \leq T(n) \leq \beta + \beta n + 6\beta(n-1) + 5\beta \sum_{k=1}^{n-1} k$$

e, como $\sum_{k=1}^{n-1} k = n(n-1)/2$*, a expressão pode ser simplificada, e obtemos*

$$2\alpha n^2 \leq -5\alpha + \frac{9}{2}\alpha n + \frac{5}{2}\alpha n^2 \leq T(n) \leq -5\beta + \frac{9}{2}\beta n + \frac{5}{2}\beta n^2 \leq 7\beta n^2,$$

porque $-5\alpha + \frac{9}{2}\alpha n + \frac{1}{2}\alpha n^2 \geq 0$ *e* $-5\beta + \frac{9}{2}\beta n \leq \frac{9}{2}\beta n^2$*, para todo* $n \geq 1$*. Portanto, provámos que* $T(n) \in \Theta(n^2)$*, isto é, que existem constantes* $C, C' \in \mathbb{R}^+$ *e* $n_0 \in \mathbb{Z}^+$ *tais que* $Cn^2 \leq T(n) \leq C'n^2$*, para todo* $n \geq n_0$*. Basta tomar,* $C = 2\alpha$ *e* $C' = 7\beta$ *e* $n_0 = 1$*.*

c) Seja $a_0, a_1, \ldots, a_{n-1}$ o estado inicial de $v[0], v[1], \ldots, v[n-1]$. Para um valor **fixo da variável** k, mas que não poderá particularizar, apresente uma condição sobre o estado das variáveis x, j e v, imediatamente após a execução do bloco de instruções 2.-6. Essa condição deve ser um invariante de ciclo e crucial para a correção do algoritmo. Justifique sucintamente que se trata de um invariante e diga como o usaria para concluir que o algoritmo resolve corretamente o problema de ordenação.

Resposta:

Sendo $a'_0 a'_1 \ldots a'_{k-1}$ *a sequência* $a_0 a_1 \ldots a_{k-1}$ *ordenada por ordem crescente, podemos descrever o estado das variáveis no ponto indicado como:*

$$\begin{aligned} x &= a_k \\ j &= \max(\{-1\} \cup \{i \mid 0 \leq i \leq k-1 \wedge a'_i \leq a_k\}) \\ v &= \begin{cases} a'_0 a'_1 \ldots a'_{k-1} a_k a_{k+1} \ldots a_{n-1} & \text{se } j = k-1 \\ a'_0 a'_1 \ldots a'_j a'_{j+1} a'_{j+1} a'_{j+2} \ldots a'_{k-1} a_{k+1} \ldots a_{n-1} & \text{se } 0 \leq j < k-1 \\ a'_0 a'_0 a'_1 \ldots a'_{k-1} a_{k+1} \ldots a_{n-1} & \text{se } j = -1 \end{cases} \end{aligned}$$

o que, quando $j \neq k-1$*, requereu um deslocamento de* $a'_{j+1} a'_{j+2} \ldots a'_{k-1}$ *de uma posição para a direita. Esse deslocamento tem por objetivo criar espaço para a inserção de* a_k *na posição correta (a qual satisfaz* $a'_j = v[j] \leq a_k < v[j+2] = a'_{j+1}$*, se* $j \neq -1$ *e* $j \neq k-1$*, e é sempre a posição* $j+1$ *de* v*).*

A condição sobre o estado das variáveis é preservada em cada iteração do ciclo "Para", pelo que se trata de um invariante desse ciclo. Após o bloco 2.–6., é executada a instrução 7. que coloca a_k *na posição correta. Assim, se, no início da iteração* k*, a sequência* $v[0]v[1]\ldots v[k-1]$ *corresponder a* $a_0 a_1 \ldots a_{k-1}$ *ordenada por ordem crescente (que se verifica de facto para* $k = 1$*) então, no fim da iteração* k*, a sequência* $v[0]v[1]\ldots v[k-1]v[k]$ *corresponde a* $a_0 a_1 \ldots a_{k-1} a_k$ *ordenada por ordem crescente, o que é importante na análise de* a_{k+1}*, na próxima iteração.*

Na última iteração do bloco de instruções do ciclo "Para", tem-se $k = n-1$*. Do invariante resulta que, se se inserir* a_{n-1} *na posição* $v[j+1]$ *então* $v[0]\ldots v[n-1]$ *será* $a_0 \ldots a_{n-1}$ *ordenada por ordem crescente. Como,* a_{n-1} *é colocado em* $v[j+1]$ *pela execução da instrução 7. a seguir ao bloco 2.–6., o invariante implica a correção do algoritmo.*

2. O algoritmo apresentado abaixo faz parte de um programa em que se pretende escrever os vértices de um grafo dirigido acíclico G (ou do seu transposto G^T) por ordem topológica. No início do ciclo "Enquanto", a pilha P está vazia, $GrauE[v]$ contém o grau de entrada do vértice v em G, para cada $v \in G.V$, e S é o conjunto dos $v \in G.V$ tais que $GrauE[v] = 0$.

```
Enquanto (S ≠ ∅) fazer
    v ← RetiraUmElemento(S);
    Push(P, v);
    Para cada w ∈ G.Adjs[v] fazer
        GrauE[w] ← GrauE[w] − 1;
        Se GrauE[w] = 0 então S ← S ∪ {w};
Se (opcao = 1) então ResG(P); senão ResTranspG(P);
```

a) Escreva as funções ResG(P) e ResTranspG(P) que imprimem $G.V$ segundo uma ordem topológica de G e de G^T. Justifique que estão corretas e indique a sua complexidade temporal.

Resposta:

```
RespG(P)
    Se (StackIsEmpty(P) = false) então
        v ← Pop(P);
        RespG(P);
        escrever(v);
```

```
RespTranspG(P)
    Enquanto (StackIsEmpty(P) = false) fazer
        escrever(Pop(P));
```

O algoritmo é análogo a um dos algoritmos descritos nas aulas para ordenação topológica, mas os nós ficam na pilha P por ordem inversa da ordem topológica (pois o primeiro fica na base e o último no topo). Por isso, em RespG(P)*, os nós são escritos por ordem inversa da ordem de saída de P.*

Como G^T é um DAG e, por definição, uma ordenação topológica de G por ordem inversa corresponde a uma ordenação topológica de G^T, então a ordem pela qual os vértices estão na pilha P define uma ordenação topológica de G^T. Por essa razão, RespTranspG(P) *escreve-os por ordem de saída de P.*

Ambas têm complexidade $\Theta(|P|)$, se Pop(P) *e* StackIsEmpty(P) *forem $O(1)$. No algoritmo dado, P contém todos os vértices de $G.V$, pelo que essas chamadas terão complexidade $\Theta(n)$, sendo $|G.V| = n$.*

b) A função RetiraUmElemento(S) escolhe um elemento de S à sorte, retira-o de S e retorna esse valor. Porque é que é correto escolher v assim?

Resposta:

É correto porque, se o grau de entrada de um nó v é zero, então v não tem precedentes. Por isso, as únicas restrições que podem envolver v são as que requerem que os números de ordem dos seus adjacentes sejam superiores ao seu. Assim, se definirmos o número de ordem $\sigma(v)$ de v na ordem topológica como o primeiro número de ordem ainda não atribuído, todas as restrições de precedência que envolvam v ficam satisfeitas. A colocação de v em P corresponde implicitamente à atribuição desse número de ordem.

c) Como é que a instrução $GrauE[w] \leftarrow GrauE[w] - 1$ contribui para a correção do algoritmo?

Resposta:

No início do ciclo "Enquanto", $GrauE[w] = |\{(v, w) \mid (v, w) \in G.E\}|$ e representa o número de vértices que impõem condições diretamente sobre $\sigma(w)$, condições que a ordenação terá de satisfazer.

Ao executar $GrauE[w] \leftarrow GrauE[w] - 1$, assinala-se que a precedência $\sigma(v) < \sigma(w)$, resultante do ramo (v, w), ficou satisfeita. Tal é correto porque se sabe que w já não ficará abaixo de v em P (o que quer dizer que w ficará necessariamente com um número de ordem maior ou igual a $\sigma(v) + 1$).

d) Admita que a complexidade temporal de RetiraUmElemento(S), de $S \leftarrow S \cup \{w\}$, e de Push($P, v$) é $\Theta(1)$. O que quer isso dizer? Justifique que: para DAGs com n vértices e m ramos, a complexidade do algoritmo apresentado é $\Theta(n+m)$, se $G.Adjs[v]$ for uma lista ligada.

Resposta:

"Complexidade temporal $\Theta(1)$" significa que o tempo que as operações referidas demoram está limitado inferiormente e superiormente por constantes, qualquer que seja o tamanho da instância considerada.

Numa representação de um grafo por listas de adjacências, a complexidade da descida da lista de adjacentes de v é $\Theta(1+|G.Adjs[v]|)$. Assim, podemos concluir que, para cada v retirado de S, a complexidade do bloco de instruções do ciclo "Enquanto" é $\Theta(1+|G.Adjs[v]|)$.

Se as funções que imprimem o resultado tiverem complexidade $\Theta(|P|)$, como na alínea 2a), então, para concluir que a complexidade do algoritmo é $\Theta(|G.V|+|G.E|)$, ou seja, $\Theta(n+m)$, basta justificar que cada vértice v de $G.V$ passa por S exatamente uma vez. Isso é verdade porque, no início, S contém todos os vértices que têm grau 0 e porque, para cada v que sai de S, o algoritmo atualiza o DAG implicitamente, removendo v e todos os ramos que saem de v, obtendo um DAG menor (definido apenas pelos vértices que ainda não estão ordenados, isto é, que ainda não estão em P). Para esse DAG, verifica se há novos vértices que ficaram com grau de entrada zero após a saída de v e coloca-os em S. Como qualquer DAG tem sempre algum vértice com grau zero (como se provou nas aulas), então S não poderá ficar vazio sem que todos os v tenham passado por S. Assim, a complexidade temporal até à instrução "Se (opcao = 1)..." é $\Theta(\sum_{v\in V}(1+|G.Adjs[v]|)) = \Theta(\sum_{v\in V} 1 + \sum_{v\in V}|G.Adjs[v]|) = \Theta(n+m)$.

3. Seja $G = (V, E)$ um grafo não dirigido finito, tal que $|V| = n$ e $|E| = m$, e os vértices são identificados por inteiros consecutivos a partir de 1. Pretendemos determinar *o número de vértices da componente conexa de G que tem mais vértices.*

a) Escreva (em pseudocódigo) um algoritmo com complexidade $O(m+n)$ que resolva o problema.

Resposta:

```
NumeroMaximo(G)
    maximo ← 0;
    Para cada v ∈ G.V fazer
        visitado[v] ← false;
    Para cada v ∈ G.V fazer
        Se (visitado[v] = false) então
            nvcomp ← DFS_Visit_Conta(v, G);
            Se (nvcomp > maximo) então
                maximo ← nvcomp;
    retorna maximo;
```

```
DFS_Visit_Conta(v, G)
    visitado[v] ← true;
    nv ← 1;
    Para cada w ∈ G.Adjs[v] fazer
        Se (visitado[w] = false) então
            nv ← nv + DFS_Visit_Conta(w, G);
    retorna nv;
```

A complexidade do algoritmo é $O(n+m)$ (mais precisamente, de ordem $\Theta(n+m)$) se G for representado por listas de adjacências, pois tem a mesma complexidade de DFS, uma vez que a contagem e determinação do máximo acrescenta apenas uma parcela linear em n (i.e., de ordem $\Theta(n)$).

b) Justifique sucintamente a correção do algoritmo que apresentou.

Resposta:

O algoritmo apresentado assume que visitado[.] é uma variável global e consiste numa adaptação simples do algoritmo de pesquisa em profundidade dado nas aulas. Como o grafo é **não dirigido**, *os vértices que* NumeroMaximo *visita na chamada* DFS_Visit_Conta(v, G) *são todos (e apenas) os da componente conexa de v. A função retorna o número de vértices que visitou. Cada vértice é contabilizado quando é marcado como visitado, o que acontece uma só vez. O número de vértices da componente de v substituirá o máximo encontrado anteriormente se for maior do que ele. O máximo começa por ser 0 o que é correto, pois qualquer componente terá um número de vértices não negativo.* (FIM)