MAC5711 Análise de Algoritmos

DCC–IME–USP, 15 de dezembro de 2004

Instruções:

(*i*) Esta prova contém seis questões sendo duas de um ponto e quatro de dois pontos.

(*ii*) Mencione os teoremas e propriedades usados para justificar suas afirmações.

(*iii*) Você pode utilizar como subrotina qualquer algoritmo **básico** visto em sala de aula sem reescrevê-lo, como, por exemplo, algoritmos para ordenação. No entando, você deve descrever clara e sucintamente o que o algoritmo recebe, devolve ou faz e o seu consumo de tempo. Exemplo

> *"O algoritmo* Blá-Blá-Blá *usa como subrotina o algoritmo* Ordenação-Lerda (A, n) *que recebe e rearranja um vetor* $A[1\,..\,n]$ *de modo que ele fique em ordem crescente. O consumo de tempo do algoritmo* Ordenação-Lerda *é* $O(n^n)$*."*

(*iv*) Não é permitida a consulta a livros, anotações, colegas, calculadoras, Internet, computadores ...

Duração da prova: 2 horas e 30 minutos

Questão 1 [1 ponto]
Considere o seguinte algoritmo que recebe um número natural n e devolve 0 se n é primo e um fator não-trivial de n se n não é primo.

```
Fator (n)
1  para d ← 2 até n − 1 faça
2       se resto(n/d) = 0
3            devolva d
4  devolva 0
```

Um aluno alega que o consumo de tempo deste algoritmo não é polinomial. O aluno está certo? Justifique a sua resposta.

Solução: O aluno tem toda a razão. Seja $\beta := \lceil \lg(n+1) \rceil$ o número de bits ou **tamanho** de n. O consumo de tempo do algoritmo Fator no pior caso é proporcional a n que é $\Theta(2^\beta)$.

Um algoritmo é **polinomial** se existe um constante c tal que seu consumo de tempo é $O(\langle I \rangle^c)$ para toda entrada I, onde $\langle I \rangle$ é o tamanho de I.

Questão 2 [1 ponto]
Meu manual diz que um problema "está em NP"; um aluno conclui que não existe algoritmo polinomial para o problema. O aluno está certo? Para justificar a sua resposta defina a classe NP.

Defina problema NP-completo. Existe algoritmo polinomial para algum problema NP-completo?

Solução: Não, ele está enganado. NP **não** é o mesmo que "não polinomial". Alias, todos os problemas de decisão que podem ser resolvidos em tempo polinomial estão em NP, isto é $\mathrm{P} \subseteq \mathrm{NP}$.

A classe NP é formada pelos problemas de decisão que possuem um verificador polinomial para a resposta SIM.

Um algoritmo polinomial ALG é um **verificador polinomial para a resposta** SIM de um problema Π se para toda entrada I de Π existe um objeto C, $\langle C \rangle$ é $\mathrm{O}(\langle I \rangle^c)$ para alguma constante c que só depende de Π, tal que

> a resposta a $\Pi(I)$ é SIM se e somente se $\textsc{Alg}(I, C)$ devolve SIM.

Um problema é NP-completo se ele está em NP e todo problema em NP pode ser reduzido a ele. Não se sabe se existe algoritmo polinomial para resolver algum problema NP-completo. Este é o intrigante problema matemático conhecido pelo rótulo "$\mathrm{P} \neq \mathrm{NP}$?"

Uma **redução** de um problema Π a um problema Π' é um algoritmo ALG que resolve Π usando uma subrotina hipotética $\textsc{Alg}'$ que resolve Π', de tal forma que, se $\textsc{Alg}'$ é um algoritmo polinomial, então ALG é um algoritmo polinomial.

Questão 3 [2 pontos]
Escreva um algoritmo NAIVE-STRING-MATCHER2(P, m, T, n) que recebe um padrão $P[1\,..\,m]$ **sem símbolos repetidos** e um texto $T[1\,..\,n]$ e devolve o número de vezes que P ocorre em T. O consumo de tempo do seu algoritmo deve ser $O(n)$. Justifique sua resposta.

Solução: O algoritmo abaixo é uma adaptação do algoritmo NAIVE-STRING-MATCHER visto em aula. Suponha que $T[n+1]$ contém símbolo que não ocorre em $P[1\,..\,m]$".

```
NAIVE-STRING-MATCHER2 (P, m, T, n)     ▷ supõe m > 0
 1  cont ← 0
 2  q ← 0    i ← 1
 3  enquanto i ≤ n + 1 faça
 4      se q = m então     ▷ caso 1: q = m
 5          ▷ "P ocorre em T com deslocamento i − q"
 6          cont ← cont + 1
 7          q ← 0
 8      senão se P[q + 1] = T[i] então    ▷ caso 2: q < m e P[q + 1] = T[i]
 9          q ← q + 1
10          i ← i + 1
11      senão se q = 0 então     ▷ caso 3: q = 0 e P[q + 1] ≠ T[i]
12          i ← i + 1
13      senão     ▷ caso 4: 0 < q < m e P[q + 1] ≠ T[i]
14          q ← 0
15  devolva cont
```

Correção

No início de cada iteração do enquanto, na linha 3, vale que

(i0) *cont* é o número de ocorrências de $P[1\,..\,m]$ em $T[1\,..\,i-2]$; e
(i1) $P[1\,..\,q] = T[i-q\,..\,i-1]$.

Se a relação invariante (i0) vale, então a correção do algoritmo é evidente. De fato, no início da última iteração das linhas 3–14 tem-se que $i = n+2$. Da invariante (i0) concluí-se que o valor *cont* devolvido pelo algoritmo é o número de ocorrências de $P[1\,..\,m]$ em $T[1\,..\,i-2] = T[1\,..\,n]$. Portanto, o algoritmo corretamente devolve o número de ocorrências de P em T.

A relação invariante (i1) garante que *cont* só é incrementada após o algoritmo encontrar uma ocorrência de P em T. A hipótese do padrão P não possuir símbolos repetidos é usada nos casos 1 e 4, quando q recebe zero, ou seja, "o padrão é deslocado de q posições".

Claramente (i0) e (i1) valem no início da primeira iteração. Com um certo trabalho pode-se verificar que se (i0) e (i1) valem no início de uma iteração então também valem no fim da iteração (com os papéis de alguns objetos alterados) e portanto valem no início da próxima iteração (verfique!).

Consumo de tempo

Cada iteração das linhas 3–14 consome tempo $\Theta(1)$. Juntos, os casos 2 e 3 ocorrem $n+1$ vezes já que:

- a variável i é inicializada com 1;
- i é incrementada de 1 nos casos 2 e 3 e nunca é decrementada; e
- no início da última iteração $i = n + 2$.

O número de ocorrências dos casos 1 e 4 é não superior ao número de ocorrências do caso 2, pois

- a variável q nunca é negativa;
- q é incrementada de 1 apenas no caso 2; e
- q é decrementada apenas no caso 1 (aqui usamos a hipótese de que $m > 1$) e no caso 4.

Logo, o número total de iterações das linhas 3–14 é não inferior a $n + 1$ e não superior a $(n + 1) + (n + 1) = 2n + 2$. Portanto, o consumo de tempo do algoritmo é $\Theta(n)$.

Questão 4 [2 pontos]
Suponha dado um conjunto de livros numerados de 1 a n. Suponha que o livro i tem peso $p[i]$ e que $0 < p[i] < 1$ para cada i. Considere o problema de acondicionar os livros no menor número possível de envelopes de modo que cada envelope tenha no máximo 2 livros e o peso do conteúdo de cada envelope seja no máximo 1. Escreva um algoritmo guloso MIN-ENV (p, n) que recebe um vetor $p[1\,..\,n]$ e devolve o número mínimo de envelopes. O consumo de tempo do seu algoritmo deve ser $O(n \lg n)$. Para mostrar que seu algoritmo está correto enuncie e prove as propriedades da *subestrutura ótima* e a da *escolha gulosa* apropriadas.

Solução: Seja $L = \{1, \ldots, n\}$ o conjunto de livros a serem acondicionados. Chamaremos qualquer conjunto com 1 ou 2 livros cuja soma dos pesos não ultrapasse 1 de **envelope**. Uma partição de L em envelopes será dita um **acondicionamente**.

Propriedade da subestrutura ótima. Seja $\mathcal{Z}$ um acondicionamento mínimo, isto é, um acondicionamento com um número mínimo de envelopes.

> Se E é um envelope em $\mathcal{Z}$, então $\mathcal{Z}' := \mathcal{Z} - \{E\}$ é um acondicionamento mínimo de $L' := L - E$.

Demonstração: Se existisse um acondicionamento $\mathcal{Y}$ de L' com $|\mathcal{Y}| < |\mathcal{Z}'|$, então teríamos que $\mathcal{Y} \cup \{E\}$ é um acondicionamento de L tal que $|\mathcal{Y} \cup \{E\}| < |\mathcal{Z}|$. Isto contrariaria a hipótese de $\mathcal{Z}$ ser um acondicionamento mínimo.

Propriedade da escolha gulosa. Suponha que a é um livro de peso máximo e b é um livro de peso mínimo.

> Se $p[a] + p[b] > 1$, então $\{a\}$ é um envelope em todo acondicionamento.
> Se $p[a] + p[b] \leq 1$, então existe acondicionamento mínimo que possui $\{a, b\}$.

Demonstração: A primeira afirmação é evidente. Passemos a verificar a segunda. Suponha que $\mathcal{Z}$ é um acondicionamento mínimo. Sejam A e B os envelopes de $\mathcal{Z}$ que contém a e b, respecticamente. Se $A = B$, então não há o que demonstrar. Suponha portanto que $A \neq B$. Sejam

$$A' := \{a, b\},\ B' := (A \cup B) - \{a, b\} \ \text{ e } \ \mathcal{Z}' := (\mathcal{Z} - \{A, B\}) \cup \{A', B'\}.$$

Temos que $|\mathcal{Z}'| = |\mathcal{Z}|$, $p[A'] = p[a] + p[b] < 1$ e

$$\begin{aligned} p[B'] &= p[(A \cup B) - \{a, b\}] \\ &= p[A] + p[B] - p[\{a, b\}] \\ &= p[a] + p[A - \{a\}] + p[b] + p[B - \{b\}] - p[a] - p[b] \\ &= p[A - \{a\}] + p[B - \{b\}] \\ &\leq p[A - \{a\}] + p[a] \qquad (1) \\ &= p[A] \leq 1 \end{aligned}$$

onde a desigualdade (1) vale pois $p[a]$ é o maior peso de um livro. Portanto, $\mathcal{Z}'$ é acondicionamento mínimo que contém $\{a, b\}$.

O algoritmo a seguir é inspirado nas propriedades da estrutura ótima e da escolha gulosa. O algoritmo usa como subrotina o algoritmo HeapSort (A, n) que recebe e rearranja um vetor $A[1\,..\,n]$ de modo que ele fique em ordem crescente. O consumo de tempo do algoritmo HeapSort é $\Theta(n \lg n)$.

```
Min-Env (p, n)
 0  HeapSort (p, n)    ▷ pré-processamento: p[1] ≤ ... ≤ p[n]
 1  i ← 1    f ← n    cont ← 0
 2  enquanto i ≤ f faça
 3       se p[i] + p[f] ≤ 1
 4            então i ← i + 1
 5       f ← f − 1
 6       cont ← cont + 1
 7  devolva cont
```

Correção

A correção do algoritmo segue das propriedades da escolha gulosa e da subestrutura ótima por indução no número de livros que ainda não foram acondicionados.

Consumo de tempo

linha	consumo de **todas** as execuções da linha
0	$\Theta(n \lg n)$
1	$\Theta(1)$
2	$\Theta(n)$
3	$\Theta(n)$
4	$\mathrm{O}(n)$
5	$\Theta(n)$
6	$\Theta(n)$
7	$\Theta(1)$
total	$\Theta(n \lg n) + 4\,\Theta(n) + \mathrm{O}(n) + 2\,\Theta(1) = \Theta(n \lg n)$

Logo, o consumo de tempo do algoritmo Min-Env é $\Theta(n \lg n)$.

Curiosidade. Sem a restrição de que cada envelope contenha no máximo 2 livros o problema é o conhecido Bin-packing, que é NP-difícil.

Questão 5 [2 pontos]
As operações PUSH, POP e STACK-EMPTY, empilham, desempilham e testam se uma pilha está vazia, respectivamente. O consumo de tempo de cada operação é $\Theta(1)$. Os algoritmos a seguir implementam uma fila usando duas pilhas, E e D.

```
ENQUEUE (x)
1  PUSH(D, x)

DEQUEUE ( )
1  se STACK-EMPTY (E) = VERDADEIRO
2      então se STACK-EMPTY(D) = VERDADEIRO
3          então devolva "fila está vazia"
4          senão enquanto STACK-EMPTY(D) = FALSO faça
5                  x ← POP(D)
6                  PUSH(E, x)
7  devolva POP(E)
```

Qual o consumo de tempo de uma sequência de m operações ENQUEUE e DEQUEUE? A sequência de operações é arbitrária. Justifique a sua resposta.

Solução: A análise a seguir foi feita pelo método de análise potencial.

Considere uma seqüência de m operações ENQUEUE e DEQUEUE. Seja

$$D_0, E_0 \xrightarrow{1^{\text{a}}\text{ op}} D_1, E_1 \xrightarrow{2^{\text{a}}\text{ op}} D_2, E_2 \longrightarrow \cdots \xrightarrow{i^{\text{a}}\text{ op}} D_i, E_i \longrightarrow \cdots \xrightarrow{m^{\text{a}}\text{ op}} D_m, E_m$$

a seqüência de pilhas obtidas durante as operações, onde o par D_i, E_i representam as pilhas D e E após a operação i.

O consumo de tempo da seqüência de operações é proporcional ao número de operações PUSH, POP e STACK-EMPTY realizadas. Definimos a função potencial Φ da fila como

$$\Phi(D, E) = 3n,$$

onde n é o número de objetos na pilha D. É evidente que $\Phi(D_i, E_i) \geq 0$ para $i = 0, 1, \ldots, m$.

Se a i-ésima operação é um ENQUEUE, então o custo real c_i da operação é 1. A diferença de potencial é

$$\Phi(D_i, E_i) - \Phi(D_{i-1}, E_{i-1}) = 3.$$

Portanto, o custo amortizado $\hat{c}_i$ desse ENQUEUE é

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i, E_i) - \Phi(D_{i-1}, E_{i-1}) \\ &= 1 + 3 \\ &= 4. \end{aligned} \tag{2}$$

Se a i-ésima operação é um DEQUEUE e k é o número de operações "POP(D)" realizados pelo DEQUEUE na linha 5, então o custo real c_i da operação é não superior a $3k + 4$: $k + 3$ operações STACK-EMPTY; k operações POP(D); k operações PUSH(E, x); e 1 operação POP(E). A diferença de potencial é

$$\Phi(D_i, E_i) - \Phi(D_{i-1}, E_{i-1}) = -3k.$$

Assim, o custo amortizado $\hat{c}_i$ desse DEQUEUE é

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i, E_i) - \Phi(D_{i-1}, E_{i-1}) \\
&\leq 3k + 4 + \Phi(D_i, E_i) - \Phi(D_{i-1}, E_{i-1}) \\
&= 3k + 4 - 3k \\
&= 4.
\end{aligned}$$

O custo amortizado de cada operação ENQUEUE e DEQUEUE é $\Theta(1)$ e portanto o consumo de tempo de uma seqüência de m operações é $\Theta(m)$. De fato, é claro que o consumo de tempo da seqüência é $\Omega(m)$. Por outro lado, tem-se que

$$\begin{aligned}
\sum_{i=1}^{m} c_i &= \sum_{i=1}^{m} (\hat{c}_i - \Phi(D_i, E_i) + \Phi(D_{i-1}, E_{i-1}) \\
&= \sum_{i=1}^{m} \hat{c}_i - \Phi(D_m, E_m) + \Phi(D_0, E_0) \\
&= \sum_{i=1}^{m} \hat{c}_i - \Phi(D_m, E_m) \\
&\leq \sum_{i=1}^{m} \hat{c}_i \qquad (3) \\
&\leq \sum_{i=1}^{m} 4 = 4m.
\end{aligned}$$

A desigualdade (3) é devida ao fato de que $\Phi \geq 0$.

Questão 6 [2 pontos]
Considere a estrutura de dados *disjoint-set forest* **com** o melhoramento *union by rank* e **sem** o melhoramento *path-compression*. Escreva os algoritmos MAKESET, UNION e FINDSET. Mostre que em uma seqüência de operações MAKESET, UNION e FINDSET, n das quais são MAKESET, o posto de cada nó na estrutura é no máximo $\lfloor \lg n \rfloor$ ou seja, que $rank[x] \leq \lg n$.

Solução: Aqui estão os algoritmos.

```
MAKESET (x)
1  pai[x] ← x
2  rank[x] ← 0

UNION (x, y)  ▷ com "union by rank"
1  x' ← FINDSET (x)
2  y' ← FINDSET (y)    ▷ supõe que x' ≠ y'
3  se rank[x'] > rank[y']
4      então pai[y'] ← x'
5      senão pai[x'] ← y'
6            se rank[x'] = rank[y']
7                então rank[y'] ← rank[y'] + 1

FINDSET (x)
1  se pai[x] = x
2      então devolva x
3      senão devolva FINDSET (pai[x])
```

Considere uma seqüência de operações MAKESET, UNION e FINDSET, n das quais são MAKESET. Seja

$$F_0 \overset{1^{\text{a}}\text{ op}}{\longrightarrow} F_i \overset{2^{\text{a}}\text{ op}}{\longrightarrow} F_2 \longrightarrow \cdots \overset{i^{\text{a}}\text{ op}}{\longrightarrow} F_i \longrightarrow \cdots \overset{m^{\text{a}}\text{ op}}{\longrightarrow} F_m$$

a seqüência de estruturas de dados *disjoint-set forest* obtidas durante as operações, onde F_i representa a estrutura após a operação i. Seja n_i o número de nós na estrutura F_i. Se x é um nó na estrutura F_i denotaremos por $pai_i[x]$ e $rank_i[x]$ o pai e o posto de x em F_i e por $n_i[x]$ o número de nós que são descendentes de x em F_i.

Queremos provar que para cada nó x em F_i vale que $rank_i[x] \leq \lg n_i[x]$. Para isto é suficiente mostrarmos a seguinte relação invariante:

$$\text{(i0) para cada } i \text{ e cada nó } x \text{ na estrutura } F_i \text{ vale que } \ 2^{rank_i[x]} \leq n_i[x]. \tag{4}$$

De (4) segue que $rank_i[x] \leq \lg n_i[x] \leq \lg n_i \leq \lg n$.

Mostraremos a relação invariante por indução em i. A relaçao (i0) vale trivialmente para $i = 0$, já que não há nós na estrutura F_0. Suponha que (i0) vale para $i = k - 1$, $k \geq 1$. Provaremos que (i0) vale para $i = k$. Temos três casos a considerar, dependendo da i-ésima operação.

Caso 1: a i-ésima operação é FINDSET(x)

Neste caso não há alteração no posto ou no número dos descendentes de cada nó.

Caso 2: a i-ésima operação é MAKESET(x)

Temos que $rank_k[x] = 0$ e $n_k[x] = 1 = 2^0$. Não há alteração no posto ou no número de descendentes dos demais nós.

Caso 3: a i-ésima operação é UNION(x, y)

Se $rank_{k-1}[x'] \neq rank_{k-1}[y']$, podemos supor sem perda de generalidade que $rank_{k-1}[x'] < rank_{k-1}[y']$. O nó y' é raiz de uma nova árvore (árvore em F_k que não existia em F_{k-1}) e

$$\begin{aligned} n_k[y'] &= n_{k-1}[x'] + n_{k-1}[y'] \\ &\geq 2^{rank_{k-1}[x']} + 2^{rank_{k-1}[y']} \\ &\geq 2^{rank_{k-1}[y']} \\ &= 2^{rank_k[y']} \end{aligned}$$

Para os demais nós não há alteração no posto ou no número de descendentes.

Se $rank_{k-1}[x'] = rank_{k-1}[y']$, o nó y' é raiz de uma nova árvore, e

$$\begin{aligned} n_k[y'] &= n_{k-1}[x'] + n_{k-1}[y'] \\ &\geq 2^{rank_{k-1}[x']} + 2^{rank_{k-1}[y']} \\ &= 2^{rank_{k-1}[y']+1} \\ &= 2^{rank_k[y']} \end{aligned}$$

Para os demais nós não há alteração no posto ou no número de descendentes.