MAC 338 – Anáise de Algoritmos

Gabarito parcial da Segunda Prova – 5 de junho de 2008

1. Dado um algoritmo linear "caixa-preta" que devolve a posição de uma mediana de um vetor, dê um algoritmo simples, linear, que resolve o problema do k-ésimo mínimo. Assuma que o protótipo deste algoritmo é ou **Mediana** (v, n) ou **Mediana** (v, p, r). Ao apresentar o seu algoritmo, diga o que ele faz em função dos parâmetros, justifique porque seu algoritmo produz a resposta correta, e analise cuidadosamente o tempo consumido por ele, mostrando que de fato ele é linear.

```
KÉSIMO-MÍNIMO (v, p, r, k)
 1  se p = r
 2     então devolva v[p]
 3  m ← Mediana(v, p, r)
 4  v[r] ↔ v[m]
 5  q ← Particione(v, p, r)
 6  d ← q − p + 1
 7  se k = d
 8     então devolva v[q]
 9     senão se k < d
10              então devolva KÉSIMO-MÍNIMO(v, p, q − 1, k)
11              senão devolva KÉSIMO-MÍNIMO(v, q + 1, r, k − d)
```

A recorrência que descreve o tempo consumido por esse algoritmo é $T(n) = T(\lfloor n/2 \rfloor) + \Theta(n)$, onde $n = r - p + 1$. A solução dessa recorrência é $\Theta(n)$.

Erros comuns que os alunos fizeram:
Esquecer de particionar o vetor com a mediana como pivô.
Errar nos índices usados em algum ponto.

2. Escreva uma versão recursiva com memoização do algoritmo descrito em aula e apresentado abaixo, que determina o custo de uma árvore de busca binária ótima. Quanto tempo consome a sua versão memoizada do algoritmo para calcular $c(1, n)$?

```
ABB-ÓTIMA (a, n)
1  s[0] ← 0
2  para i ← 1 até n faça
3     s[i] ← s[i−1] + a[i]
4  para i ← 1 até n+1 faça
5     c[i][i − 1] ← 0
6     para j ← i até n faça
7        c[i][j] ← −1
8  devolva ABB-ÓTIMA-REC (c, s, 1, n)
```

```
ABB-ÓTIMA-REC (c, s, i, j)
1  se c[i][j] ≠ −1
2     então devolva c[i][j]
3  c[i][j] ← ABB-ÓTIMA-REC(c, s, i + 1, j)
4  para k ← i+1 até j faça
5     t ← ABB-ÓTIMA-REC(c, s, i, k − 1) + ABB-ÓTIMA-REC(c, s, k + 1, j)
6     se c[i][j] > t
7        senão c[i][j] ← t
8  c[i][j] ← c[i][j] + s[j] − s[i − 1]
9  devolva c[i][j]
```

O tempo consumido pela versão memoizada do algoritmo é o mesmo do algoritmo não memoizado, ou seja, $\Theta(n^3)$. De fato, descontada as chamadas recursivas, o algoritmo acima consome tempo $O(n)$. O número de chamadas recursivas diferentes é $O(n^2)$. Chamadas executadas pela primeira vez consomem $O(n)$ e chamadas repetidas consomem $O(1)$. Com isso, no pior caso, o tempo consumido é $O(n^3)$. Em um exemplo em que todas as entradas da matriz c são preenchidas, o consumo será de $\Theta(n^3)$, pois, por exemplo, toda primeira chamada em que $j - i \geq n/2$ consome $\Theta(n)$, e existe um número $\Theta(n^2)$ de entradas assim.

Erros comuns que os alunos fizeram:
Alguns alunos afirmavam que o tempo do algoritmo memoizado era $\Theta(n^2)$ alegando que o número de posições da matriz c é $\Theta(n^2)$. Mas é preciso considerar o tempo que o algoritmo gasta para preencher uma entrada da matriz na conta... Não basta olhar o número de posições da matriz...

3. Dado n e uma cadeia de n caracteres $s[1\,..\,n]$ que você acredita ser um texto corrompido, em que toda a pontuação foi removida (de modo que pareça com alguma coisa assim... "eraumavezumgatoxadrez..."). Você deseja reconstruir o documento, usando um dicionário, que está disponível na forma de uma função booleana `dict(.)`: para cada cadeia de caracteres w,

$$\texttt{dict(.)} \;=\; \begin{cases} \texttt{true} & \text{se } w \text{ é uma palavra válida} \\ \texttt{false} & \text{caso contrário.} \end{cases}$$

Escreva um algoritmo de programação dinâmica que, dado n e uma cadeia de caracteres $s[1\,..\,n]$, determina se s pode ser reconstituída como uma seqüência de palavras válidas. O seu algoritmo deve consumir tempo $O(n^2)$. Justifique porque ele funciona (por exemplo explicando a validade da recorrência de onde ele foi derivado) e porque o seu consumo de tempo é $O(n^2)$.

O enunciado deveria ter dito adicionalmente que poderíamos assumir que a função `dict(.)` consome tempo $O(1)$ na análise do algoritmo.

Seja $f(i)$ a seguinte função:

$$f(i) \;=\; \begin{cases} 1 & \text{se } s[1\,..\,i] \text{ é válida} \\ 0 & \text{caso contrário,} \end{cases}$$

definida para $0 \leq i \leq n$. Observe que $f(i)$ satisfaz a seguinte recorrência:

$$f(i) \;=\; \begin{cases} 1 & \text{se } i = 0 \\ \max\{f(j) : 0 \leq j < i \text{ e } \texttt{dict}(s[j+1\,..\,i])\} & \text{se } i \geq 1. \end{cases}$$

Dessa recorrência, é fácil derivar um algoritmo que consome tempo $O(n^2)$ para resolver o problema.

```
VÁLIDA (s, n)
1  f[0] ← true
2  para i ← 1 até n faça
3     f[i] ← false
4     j ← i − 1
5     enquanto f[i] = false e j ≥ 0 faça
6        se dict(s[j + 1 .. i])
7           entãof[i] ← f[j]
8        j ← j − 1
9  devolva f[n]
```

É fácil ver que o algoritmo acima consome $O(n^2)$. Na verdade, consome $\Theta(n^2)$ no pior caso.

Erros comuns que os alunos fizeram:
Vários alunos fizeram uma recorrência para uma função $g(i, j)$ que valia 1 se e somente se $s[i\,..\,j]$ é válida. Tal recorrência de fato também leva a um algoritmo de PD para resolver o problema, porém o algoritmo é (se bem implementado) $\Theta(n^3)$ no pior caso. Isso porque, de novo, o número de entradas da matriz construída é $\Theta(n^2)$ e, para cada entrada, consome-se $O(n)$ (com $\Theta(n)$ para mais da metade das entradas da matriz). No entanto, vários alunos, após apresentar o algoritmo derivado desta outra recorrência, afirmaram erroneamente que seu algoritmo era $O(n^2)$...

4. Seja $x_1, x_2, \ldots, x_n$ uma seqüência de números, onde n é par. Um *pareamento* de $x_1, x_2, \ldots, x_n$ é uma partição do (multi)conjunto $\{x_1, x_2, \ldots, x_n\}$ em pares. Se P é um pareamento de $x_1, x_2, \ldots, x_n$, então a *altura* de P é o valor $\max x_i + x_j : \{x_i, x_j\}$ é um par de P. Um pareamento de $x_1, x_2, \ldots, x_n$ é ótimo se tem altura mínima.

   Escreva um algoritmo que, dado n e os números $x_1, x_2, \ldots, x_n$, encontra um pareamento ótimo de $x_1, x_2, \ldots, x_n$. Seu algoritmo deve consumir tempo $O(n \lg n)$. Justifique que ele de fato produz uma resposta correta, e que o seu consumo de tempo é de fato $O(n \lg n)$.

```
PAREAMENTO (x, n)
1  Ordena(x, n)                ▷ em ordem crescente
2  para i ← 1 até n/2 faça
3     P[i] ← {x[i], x[n − i + 1]}
4  devolva P
```

   É fácil ver que o algoritmo consome tempo $O(n \lg n)$. Resta justificar porque ele produz um pareamente ótimo.

   Tome um pareamento ótimo $P*$ que inclua pares $P[1], \ldots, P[i]$ para i o maior possível. Se $i = n/2$, não há nada a provar: $P = P*$ e portanto P é um pareamento ótimo. Se $i < n/2$, então considere o pareamente P', derivado de $P*$ da seguinte maneira. Em $P*$ os elementos $x[i+1]$ e $x[n-(i+1)+1]$ não formam um par. Então sejam j e k tais que, em $P*$, temos os pares $\{x[i+1], x[j]\}$ e $\{x[n-(i+1)+1], x[k]\}$. Seja P' o pareamento que coincide com $P*$ exceto pela troca entre $x[j]$ e $x[n-(i+1)+1]$ nos pares acima. Qual é a altura do pareamento P'?

   Observe que $i+1 < j < n-(i+1)+1$, logo $x[j] \leq x[n-(i+1)+1]$. Ou seja, a soma do segundo par (para onde foi $x[j]$) ou ficou igual ou diminuiu. Em fórmulas, $x[j]+x[k] \leq x[n-(i+1)+1]+x[k]$. Por outro lado, como $x[i+1] \leq x[k]$, temos também que $x[i+1]+x[n-(i+1)+1] \leq x[k]+x[n-(i+1)+1]$. Ou seja, com certeza a altura de P' é menor ou igual á altura de $P*$. Mas como $P*$ é um pareamento ótimo, essas alturas são iguais e P' é também um pareamento ótimo. Porém P' tem um par a mais em comum com P (coincidiria com P até $i+1$ pelo menos), uma contradição a escolha de $P*$. Ou seja, esse caso não ocorre. Ocorre apenas o caso em que $i = n/2$, e portanto P é um pareamento ótimo.

Erros comuns que os alunos fizeram:
A maioria dos alunos descreveu o algoritmo correto. Mas apenas um aluno chegou perto de escrever um argumento correto de que o algoritmo de fato devolve um pareamento ótimo. Alguns dão um argumento que significa que o pareamento devolvido é um "ótimo local", no sentido de que trocando dois elementos em um par não se obtem um pareamento melhor. Mas esse fato isolado não prova que o pareamento é ótimo. É preciso saber usar isso dentro de um argumento que compara um pareamento ótimo com o do algoritmo. Aqui, é importante observar que há casos em que um ótimo "local" não é um ótimo global, por isso tal argumento isolado não serve como prova.

Alguns alunos também argumentaram que, num pareamento ótimo, o menor sempre forma um par com o maior, mas isso não é verdade... Pense por exemplo em $x = [1, 2, 3, 30, 31, 48, 49, 50]$. O seguinte pareamento é ótimo: $\{\{1, 48\}, \{2, 49\}, \{3, 50\}, \{30, 31\}\}$. Ou seja, existem pareamentos ótimos que não formam um par com o menor e o maior elemento.