Gabarito da Prova 3

`MAC338 - 4/7/2000`

Resposta da questão 1.

(i)

Se $n^\alpha = O(n^\beta)$ então existem constantes c, n_0 tais que

$$n^\alpha \leq cn^\beta \quad \text{para todo } n \geq n_0.$$

Se $\alpha > \beta$ então existiria $n_1 \geq n_0$ tal que $n^{\alpha-\beta} > c$, o que seria uma contradição. Logo, $\alpha \leq \beta$.

Por outro lado, se $\alpha \leq \beta$ então $n^\alpha \leq n^\beta$ para todo $n \geq 0$. Portanto, $n^\alpha = O(n^\beta)$.

(ii)

Vamos mostrar que $n^{\alpha_1}(\log n)^{\alpha_2} = O(n^{\beta_1}(\log n)^{\beta_2})$ se e somente se vale uma das seguintes condições:

- $\alpha_1 < \beta_1$ ou,
- $\alpha_1 = \beta_1$ e $\alpha_2 \leq \beta_2$.

Necessidade. Vamos usar o seguinte fato: se $\epsilon > 0$ então para todo k:

$$(\log n)^k = O(n^\epsilon) \text{ mas } n^\epsilon \neq O((\log n)^k).$$

Suponha que existem constantes c, n_0 tais que

$$n^{\alpha_1}(\log n)^{\alpha_2} \leq cn^{\beta_1}(\log n)^{\beta_2} \quad \text{para todo } n \geq n_0.$$

Se $\alpha_1 > \beta_1$ então pelo fato mencionado acima existe $n_1 \geq n_0$ tal que $n^{\alpha_1-\beta_1} > c(\log n)^{\beta_2-\beta_2}$, o que é uma contradição. Então $\alpha_1 \leq \beta_1$. Se $\alpha_1 < \beta_1$ não há nada a provar. Suponha que $\alpha_1 = \beta_1$ e $\alpha_2 > \beta_2$. Então existe $n_2 > n_0$ tal que $(\log n)^{\alpha_2-\beta_2} > c$, o que leva a uma contradição. Portanto, $\alpha_1 = \beta_1$ e $\alpha_2 \leq \beta_2$.

Suficiência. Se $\alpha_1 < \beta_1$, então existem constantes c, n_0 tais que

$$(\log n)^{\alpha_2-\beta_2} \leq cn^{\beta_1-\alpha_1} \quad \text{para todo } n \geq n_0.$$

Se $\alpha_1 = \beta_1$ e $\alpha_2 \leq \beta_2$ então

$$(\log n)^{\alpha_2} \leq (\log n)^{\beta_2} \quad \text{para todo } n \geq 2.$$

Em ambos os casos, $n^{\alpha_1}(\log n)^{\alpha_2} = O(n^{\beta_1}(\log n)^{\beta_2})$.

(iii)

Se $p_k = \Theta(k \log k)$ então existem constantes a, b, k_0 tais que

$$ak \log k \leq p_k \leq bk \log k \quad \text{para todo } k \geq k_0.$$

Suponha que $\pi(n) = k$, com $k \geq k_0$. Assim, $p_k \leq n$ mas $n < p_{k+1}$. Temos que

$$\begin{aligned} n < p_{k+1} &\leq b(k+1)\log(k+1) \\ &\leq b(2k)\log(2k) \\ &\leq 2bk(\log k + 1) \text{ (log na base 2)} \\ &= 2bk \log k + 2bk \\ &\leq Bk \log k, \end{aligned}$$

onde $B = 4b$. Assim, $n \leq B\pi(n)\log\pi(n)$ para $n \geq n_0 = p_{k_0}$, e portanto

$$\pi(n) \geq \frac{n}{B\log\pi(n)} \geq \frac{n}{B\log n} \quad \text{para todo } n \geq n_0,$$

ou seja, $\pi(n) = O(n/\log n)$.

Por outro lado, como $n \leq Bk\log k$, $\pi(n) = k$, temos que

$$\log n \leq \log B + \log k + \log\log k,$$

e portanto,

$$\log k \geq \log n - \log\log n - \log B \geq C\log n,$$

para alguma constante C e para todo $n \geq n_0$. Assim,

$$\pi(n) \leq \frac{n}{a\log\pi(n)} \leq \frac{n}{aC\log n} \quad \text{para todo } n \geq n_0.$$

Logo, $\pi(n) = \Theta(n/\log n)$.

Resposta da questão 2.

Um elemento x que é resposta do *Problema da Seleção* é chamado de k-ésimo de `x`. Para facilitar as respostas dos itens vamos supor que a seqüência **x** está armazenada em um vetor `x[]` (início em 1 e final em n).

(i)

Execute um algoritmo de ordenação com complexidade de pior caso $O(n\log n)$ sobre o vetor `x[]`, e devolva `x[k]`. Poderia-se usar por exemplo o heapsort ou o mergesort.

(ii)

Para resolver este item, vamos supor que temos à disposição uma função `partition(x,l,r)` como no algoritmo quicksort, que recebe um vetor `x[]` com início em l e final em r e devolve um índice i, $l \leq i \leq r$ e rearranja o vetor de modo que $\texttt{x}[j] \leq \texttt{x}[i]$, para $l \leq j \leq i$, e $\texttt{x}[j] \geq \texttt{x}[i]$, para $i < j \leq r$.

```
Select(int x[], int l, int r, int k)
{
   if (l == r) return x[r];
   i = partition(x,l,r);
   q = i-l+1;
   if (k == q) return x[i];
   if (k < q)
      return Select(x,l,i-1,k);
   else
      return Select(x,i+1,r,k-q);
}
```

O algoritmo funciona do seguinte modo. A cada chamada recursiva de `Select(x,l,r,k)` temos $l \leq r$ e $k \leq r - l + 1$. A função `partition` é executada e devolve um índice i. A variável q indica o número de elementos no vetor entre l e i. Se $k = q$ então devolvemos $\texttt{x}[i]$, ele é o k-ésimo de `x`. Senão, analisamos os tamanhos dos subvetores determinados por i.

Se o subvetor de `x` que começa em l e termina em $i - 1$ possui pelo menos k elementos, chamamos recursivamente o algoritmo para encontrar o k-ésimo dele, que é o k-ésimo do vetor `x`. Senão, chamamos recursivamente o algoritmo para encontrar o $(k-q)$-ésimo elemento do subvetor que começa em $i+1$ e termina em r, que é o k-ésimo de `x`.

(iii)

Denote por $T(n)$ a complexidade de caso médio do algoritmo `Select`. O algoritmo invoca a função `partition` que divide o vetor em dois subvetores de tamanho i e $n-1-i$, onde $n := r-l+1$. Os possíveis valores de i são $0, 1, 2, \ldots, n-1$. Podemos supor, já que a distribuição é uniforme, que a probabilidade de i ser algum desses valores é $1/n$. Além disso, o tempo de execução da função `partition` é $O(n)$. Então

$$T(n) \leq \frac{1}{n}\sum_{i=0}^{n-1} T(i) + O(n).$$

Vamos provar por indução que $T(n) \leq cn$, onde $c > 0$ é uma constante que não depende de n. Então

$$T(n) \leq \frac{1}{n}\sum_{i=1}^{n-1} ci + O(n) = \frac{c}{n}\binom{n}{2} + O(n) = \frac{1}{2}c(n-1) + O(n) \leq cn,$$

onde a última desigualdade segue se tomarmos c suficientemente grande em relação à constante escondida na parcela $O(n)$. Logo, $T(n) = O(n)$.

Resposta da questão 3.

(i)

O número de vezes que a atribuição `a[x][j] = 1` é executada é no máximo n^3. Este limite é o melhor possível como pode ser visto no caso em que todas as entradas da matriz de adjacência são iguais a 1.

(ii)

O trecho de código acima calcula o chamado *fecho transitivo* do grafo orientado dado pela matriz de adjacência A (não era necessário saber o nome). O fecho transitivo F de G é o grafo com conjunto de vértices $V(G)$ e com a seguinte propriedade: se existe um caminho orientado de i a j em G, então existe um arco (i, j) em F. Após a execução do código descrito, a matriz A é a matriz de adjacência de F.

Resposta da questão 4.

(i)

A função `fixDown()` é usada na manipulação (ou construção) de um heap representado pelo vetor $a[]$, como início em 1 e final em N. Os filhos de um elemento $a[j]$ são $a[2j]$ e $a[2j+1]$, se $2j < N$; se $2j = N$ então o único filho de $a[j]$ é $a[N]$, e se $2j > N$ então $a[j]$ não tem filhos. Dizemos que a é um *heap em j* se $a[j]$ é menor ou igual aos seus filhos.

Na chamada da função `fixDown()`, as seguintes condições devem estar satisfeitas:

1. $1 \leq k \leq N$;
2. Para cada descendente próprio $a[j]$ de $a[k]$ temos que a é um heap em j.

O efeito da execução de `fixDown()` é fazer com que a se torne um heap em k sem violar esta propriedade dos seus descendentes.

(ii)

A *altura* $h(x)$ de um elemento x em uma árvore pode ser definida recursivamente da seguinte forma: se x é uma folha então $h(x) = 0$; senão $h(x) = \max\{h(y) + 1 : y \text{ é filho de } x\}$. Em outras palavras, é distância máxima de x a uma folha usando caminhos que *não sobem* na árvore. Considere por exemplo uma árvore binária correspondente a um heap a de 12 elementos: Então

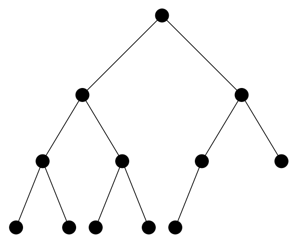

Figura 1: Árvore binária completa de 12 elementos.

- $a[1]$ tem altura 3;
- $a[2], a[3]$ têm altura 2;
- $a[4], a[5], a[6]$ têm altura 1;
- os demais elementos têm altura 0.

(iii)

O número máximo de vezes que `exch(a[k], a[j])` é executado é no máximo a altura de $a[k]$ no heap (pensando na árvore binária).

(iv)

O trecho de código correspondente à construção de um heap é o seguinte:

```
for (i = N/2; i > 0; i--)
   fixDown(a,i,N);
```

onde $n = N$. Pelo item anterior, o número de execuções de `exch(a[k], a[j])` é no máximo a soma $f(n)$ das alturas dos elementos da árvore binária completa com n nós (note que para $i > N/2$, a altura de $a[i]$ é zero). Assim, o tempo de construção do heap é $O(f(n))$.

(v)

Claramente $f(1) = 0$ e $f(2) = f(3) = f(1) + 1$. Seja T uma árvore binária completa de n nós. Sabemos que T possui exatamente $\lceil n/2 \rceil$ folhas. Seja T' a árvore obtida de T removendo-se essas folhas. É fácil ver que T' é uma árvore binária completa com $\lfloor n/2 \rfloor$ nós. A altura de cada um destes nós em T' é exatamente um a menos da altura dele em T. Portanto, $f(n) = f(\lfloor n/2 \rfloor) + \lfloor n/2 \rfloor$.

(vi)

Vamos provar por indução em n. Temos que $f(1) = 1 - b(1)$. Suponha que $n \geq 2$. Então,

$$f(n) = f(\lfloor n/2 \rfloor) + \lfloor n/2 \rfloor = \lfloor n/2 \rfloor - b(\lfloor n/2 \rfloor) + \lfloor n/2 \rfloor = 2\lfloor n/2 \rfloor - b(\lfloor n/2 \rfloor).$$

Se n for par então segue que $2\lfloor n/2 \rfloor = n$ e $b(\lfloor n/2 \rfloor) = b(n)$. Se n for ímpar então $2\lfloor n/2 \rfloor = n - 1$ e $b(\lfloor n/2 \rfloor) = b(n) - 1$. Em ambos os casos, $f(n) = n - b(n)$.

Segue do item (iv) que podemos construir um heap de n elementos com no máximo $f(n) = n - b(n) \leq n$ trocas.