Prova Substitutiva — PMR3201 1o. semestre 2016

Prof. Thiago Martins

Instruções: Escreva o nome e o número USP na folha de papel almaço. Numere cada página. Indique o total de páginas na primeira página. Os códigos fornecidos na seção "Códigos-fonte de apoio" podem ser referenciados em qualquer resposta sem necessidade de reprodução.

1. (2,0 pontos) Uma sequência A é dita *subsequência* de outra sequência B se é possível transformar B em A removendo-se elementos mantendo-se a ordem original. Por exemplo, a sequência $\{2, 5, 1\}$ é subsequência da sequência $0, 2, 1, 1, 5, 2, 1$. Por outro lado, a sequência $\{1, 2, 2\}$ *não é* subsequência da sequência $2, 0, 1, 2$. Escreva uma função em Java que determina se uma sequência é subsequência de outra. Use a seguinte assinatura:

   ```
   static boolean checaSubsequencia(int[] x, int[] y)
   ```

   Esta função deve testar se a subsequência no vetor `x` é subsequência do vetor `y`. Seja N o tamanho do vetor `y`. Um algoritmo $\mathcal{O}(N)$ vale 2,0 pontos, algoritmos com complexidade pior valem 1,0 pontos.

   Resposta: A solução é manter dois índices, um para os elementos em `x`, outro para os elementos em `y`. Os elementos em `y` são inspecionados sequencialmente. A cada vez que o elemento em `x` é igual ao elemento em `y`, o índice de `x` é incrementado. Se por este processo chega-se ao final da sequência em `x`, retorna-se verdadeiro, falso caso contrário.

   ```
   static boolean checaSubsequencia(int[] x, int[] y) {
       int i, j;
       for(i = j = 0; i<y.length && j < x.length; i++)
           if(x[j]==y[i]) j++;
       return j==x.length;
   }
   ```

2. (2,0 pontos) O código abaixo busca por um elemento em um vetor *não*-ordenado de forma recursiva:

   ```
    1  static int buscaElemento(int x, int[] y, int e, int d) {
    2      if(e>d) return -1;
    3      if(e==d) {
    4          if(y[e]==x) return e;
    5          else return -1;
    6      }
    7      int m = (e+d)/2;
    8      int e = buscaElemento(x, y, e, m);
    9      if(e!=-1) return e;
   10      return buscaElemento(x, y, m+1, d);
   11  }
   ```

   O parâmetro `x` é o elemento a ser procurado, `y` é o vetor no qual o elemento deve ser buscado, `e` é o limite inferior da posição no vetor no qual a busca deve ser feita e `d` é o limite superior. Caso a função encontre o elemento, retorna o índice no vetor `y` onde ele se encontra. Caso contrário, retorna -1. A função é recursiva: ela divide o vetor em duas metades e chama a si mesma em cada metade até encontrar um vetor de tamanho unitário ou nulo.

(a) (0,5 pontos) Mostre que o algoritmo está correto.

> **Resposta:** Trata-se de um algoritmo recursivo. O caso base é o caso de um vetor nulo ou unitário. No vetor nulo o algoritmo retorna -1, pois trivialmente o elemento buscado não está presente. No vetor unitário o algoritmo retorna o índice inspecionado caso o elemento lá presente corresponda ao elemento buscado ou -1 caso contrário.
>
> Assim o caso base está correto. No caso de vetores maiores o algoritmo divide o vetor em duas partes uma de `e` até `m`, outra de `m` + 1 até `d`. Como estas duas partes encopassam todo o vetor, se o elemento buscado está presente no vetor, ele necessariamente está em uma destas partes. Ademais, estes vetores têm comprimento não-negativo e estritamente menores que o vetor original. Supondo assim que as chamadas recursivas na linha 8 e 10 retornam o valor correto, o algoritmo está correto. Ora, mas foi visto que para o caso base elas estão corretas. Por outro lado o caso base é *sempre* atingido. Assim o algoritmo está correto.

(b) (1,0 pontos) Encontre a complexidade do algoritmo e compare-a com um algoritmo de busca sequencial.

> **Resposta:** O algoritmo faz duas chamadas recursivas em problemas com metade do tamanho do original e operações em tempo *constante* (ou seja, $\mathcal{O}(1)$).
>
> Em notação do *Master Theorem*, $A = 2$, $B = 2$ e $L = 0$. Assim, $A > B^L$, de modo que a solução é $\mathcal{O}\left(N^{\log_2 2}\right)$ ou seja, $\mathcal{O}(N)$.
>
> O algoritmo de busca sequencial inspeciona cada elemento do vetor um-a-um, com complexidade $\mathcal{O}(N)$. Deste modo, os algoritmos *são equivalentes* em termos de complexidade assintótica de número de operações (mais do que isso, é possível mostrar que eles fazem comparações na mesma ordem e retornam sempre o mesmo resultado, mesmo quando há elementos repetidos).

(c) (0,5 pontos) Compare o código com o algoritmo de busca sequencial em termos de *uso de memória*.

> **Resposta:** O algoritmo divide sucessivamente o vetor em 2, fazendo um nível de chamada recursiva por divisão. Assim ele usa uma ordem de $\mathcal{O}(\log N)$ de memória de pilha por execução. Por outro lado, uma implementação trivial do algoritmo de busca sequencial usa memória *constante*. Assim este algoritmo recursivo é menos eficiente em termos de uso de memória.

3. (2,0 pontos) O código abaixo faz uma busca similar a do item anterior, no entanto desta vez o vetor no qual a busca se dará é *ordenado*:

,

```
public static int buscaBinaria(int x, int y[], int e, int d) {
    if(e>d) return -1;
    int m = (e+d)/2;
    if(y[m]==x) return m;
    if(y[m]>x) return buscaBinaria(x, y, e, m-1);
    return buscaBinaria(x, y, m+1, d);
}
```

Novamente, o parâmetro `x` é o elemento a ser procurado, `y` é o vetor no qual o elemento deve ser buscado, `e` é o limite inferior da posição no vetor no qual a busca deve ser feita e `d` é o limite superior. Caso a função encontre o elemento, retorna o índice no vetor `y` onde ele se encontra. Caso contrário, retorna -1.

Esta função é uma implementação recursiva do conhecido algoritmo de busca binária, que encontra elementos em vetores ordenados com complexidade $\mathcal{O}(\log N)$.

Uma complexidade similar pode ser obtida em árvores binárias de busca, mas somente com árvores *balanceadas*.

Escreva uma função em java que, dado um vetor ordenado, constrói uma árvore binária de busca de forma que a sequência de comparações feitas nesta árvore é a mesma que a feita pela função `buscaBinaria` acima. Use a seguinte assinatura:

```
static NoArvoreBinaria montaArvoreDeBusca(int x[])
```

Onde `x` é o vetor ordenado. A função deve retornar um elemento `NoArvoreBinaria` com a raiz da árvore construída.

Resposta: A idéia é inserir os valores na mesma ordem em que eles seriam consultados em uma busca binária. A inserção nesta ordem garante a preservação da ordem na busca. O desafio é gerar *todas* as ordens de busca para *todos* os elementos no vetor. Mas a forma recursiva presente no enunciado oferece uma solução simples para este problema: ignorar o resultado da comparação e gerar buscas recursivas para os dois lados do vetor.

,

```
static NoArvoreBinaria montaArvoreDeBusca(int x[], int e, int d) {
    if(e>d) return null;
    int m = (e+d)/2;
    NoArvoreBinaria no = new NoArvoreBinaria();
    no.valor = x[m];
    no.esquerda = montaArvoreDeBusca(x, e, m-1);
    no.direita = montaArvoreDeBusca(x, m+1, d);
    return no;
}
```

4. (2,0 pontos) Um heap binário é uma árvore binária completa (todos os níveis exceto o último cheios) preenchido da esquerda para a direita na qual cada nó é maior ou igual a seus filhos. Heaps binários são armazenados em vetores, resultantes da varredura da árvore binária *em largura*. Dada a rígida estrutura, é possível converter um vetor em árvore e vice-versa.

   Há dois algoritmos relevantes para alterações em um heap binário implementados nas seguintes funções:

   `static void FixHeapDown(int a[], int start, int end)` corrige um heap binário no qual um nó é menor ou igual ao seu pai mas possivelmente menor do que os seus filhos. O nó é empurrado "para baixo" no heap até encontrar uma posição adequada. O parâmetro `a` é o vetor que contém o heap, o parâmetro `start` indica o nó cuja posição está incorreta e o parâmetro `end` indica a última posição do heap no vetor.

   `static void FixHeapUp(int[] a, int fim)` corrige um heap binário no qual o nó final é possivelmente maior do que o seu pai. O nó é empurrado "para cima" no heap até encontrar uma posição adequada. O parâmetro `a` é o vetor que contém o heap, e o parâmetro `end` indica a última posição do heap no vetor, onde está o elemento possivelmente errado.

   Escreva uma função em java que remove um elemento *arbitrário* de um heap mantendo a sua estrutura (mas naturalmente reduzindo o seu tamanho). Use a seguinte assinatura:

```
static int removeFromHeap(int[] a, int end, int pos)
```

   Onde `a` é o vetor que contém o heap binário, `end` é a posição do fim do heap no vetor e `pos` é o índice no vetor, entre 0 e `end`, do elemento a ser removido. A função deve retornar a nova posição do final do heap

(ou seja, `end`-1). A função deve operar com uma complexidade $\mathcal{O}(\log N)$ ou melhor (complexidades piores valem 1,0 pontos).

Resposta: A última posição no heap pode ser removida sem prejuízo da estrutura do mesmo. Deseja-se no entanto remover um elemento *arbitrário*. Ora, mas ao se trocar o último elemento de posição com o elemento que deseja-se substituir, gera-se um heap com um elemento potencialmente "errado", menor do que seus pais mas possivelmente menor que seus filhos, exatamente a solução que pode ser corrigida por `FixHeapDown`, que o faz na complexidade desejada. O código é:

,

```
static int removeFromHeap(int[] a, int end, int pos) {
     int temp = a[pos];
     a[pos] = a[end];
     a[end] = temp;
     end--;
     FixHeapDown(a, pos, end);
     return end;
}
```

5. (2,0 pontos) A classe `Pilha` implementa uma pilha de inteiros por meio de lista ligada. O método `PilhaAr()` constrói uma nova pilha. O método `push(int x)` empilha o valor `x`. O método `int pop()` retira o último elemento empilhado e retorna-o ou lança a exceção `ErroPilhaVazia` caso a pilha esteja vazia. O método `int top()` retorna o elemento no topo da pilha *sem modificá-la* ou lança a exceção `ErroPilhaVazia` caso a pilha esteja vazia. o método `PilhaVazia()` retorna verdadeiro se a pilha está vazia, falso caso contrário. A classe realiza todas estas operações em tempo constante.

   Implemente uma nova classe `PilhaPar` que implementa *todos* os métodos descritos acima (construtor, `push`, `pop`, `top` e `PilhaVazia`). Além disso ela deve implementar o método `int par()` que retorna o mais recente elemento *par* a ser empilhado *sem modificar o estado da pilha*. Estas operações devem ser realizadas em tempo constante.

Resposta: A solução é manter *duas* pilhas, uma geral e uma apenas para elementos pares. Quando o método `push` é invocado com um elemento par, ele é empilhado nas duas pilhas. O método `pop` é incialmente chamado na pilha geral. Se o elemento retornado for par, ele é invodado também na pilha de elementos pares. O elemento par é retornado pela função `top` da pilha de elementos pares.

,

```
public class PilhaPar {
  Pilha p1, p2;

  PilhaPar() {
      p1 = new Pilha();
      p2 = new Pilha();
  }

  public void push(int x) {
      p1.push(x);
      if(x%2==0) p2.push(x);
  }

 public int pop() {
```

```
        int x = p1.pop();
        if(x%2==0) p2.pop();
        return x;
    }

    public int par() {
        return p2.top();
    }

    public int top() {
        return p1.top();
    }

    public boolena PilhaVazia() {
        return p1.PilhaVazia();
    }
}
```

Formulário

Solução de equações de recorrência pelo *Master Theorem*:

A equação:

$$T(N) = A \cdot T\left(\frac{N}{B}\right) + cN^L,$$

com $T(1) = \mathcal{O}(1)$, tem solução

$$T(N) = \begin{cases} \mathcal{O}\left(N^{\log_B A}\right) & \text{se } A > B^L \\ \mathcal{O}\left(N^L \log N\right) & \text{se } A = B^L \\ \mathcal{O}\left(N^L\right) & \text{se } A < B^L \end{cases}$$

Códigos-fonte de apoio

Classe `Pilha`

```
public class Pilha {
   public class ErroPilhaVazia extends java.lang.RuntimeException {}

   private NoListaLigada topo;

   public Pilha() {
   }

   public void push(int x) {
       this.topo = new NoListaLigada(x, this.topo);
   }

   public int pop() {
      if (topo == null) throw new ErroPilhaVazia();
      int x = topo.value;
      topo = topo.next;
      return x;
   }

   public int top() {
      if (topo==null) throw new ErroPilhaVazia();
      return topo.value;
   }

   public boolean PilhaVazia() {
       return topo==null;
   }
}
```

Classe `NoListaLigada`:

```
public class NoListaLigada {
    public int value;
    public NoListaLigada next;

    NoListaLigada(int value, NoListaLigada next) {
        this.value = value; this.next = next;
    }
}
```

Classe `NoArvoreBinaria`

```
public class NoArvoreBinaria {
  public int valor;
  public NoArvoreBinaria esquerda, direita;
}
```

Função `FixHeapDown`:

```
    static void FixHeapDown(int a[], int start, int end) {
        int pai = start;
        int filho = 2*pai+1;
        int maior;
        while(filho<=end) {
            maior = pai;
            if(a[pai]<a[filho])
                maior = filho;
            if(filho+1<=end && a[maior]<a[filho+1])
                maior = filho+1;
            if(maior==pai) return;
            int aux = a[pai];
            a[pai] = a[maior];
            a[maior] = aux;
            pai = maior;
            filho = 2*pai+1;
        }
    }
```

Função `FixHeapUp`:

```
    static void FixHeapUp(int[] a, int fim) {
        int pai = (fim-1)/2;
        while(pai!=fim) {
            if(a[pai]<a[fim]) {
                int aux = a[pai];
                a[pai] = a[fim];
                a[fim] = aux;
                fim = pai;
                pai = (fim-1)/2;
            } else return;
        }
    }
```