Métodos de Programação II

LESI / LMCC

15 de Junho 2004

Questão 1 [*Análise de correcção de algoritmos*]
Considere o problema de se encontrar o maior número de valores iguais num **array ordenado**, e um algoritmo que encontra a solução percorrendo o array apenas uma vez, armazenando uma estimativa do valor correcto numa variável `size`.

```
int maiorRepet(int A[], int n) {
   int i;
   int size;

   // Outras declaracoes

   if (n<=0) return 0;
   else if (n==1) return 1;
   else {
     // Inicializacoes

     for (i=1; i<n; i++) {
        // Teste e actualizacao da variavel size
     }
   }
}
```

1. (a) Que propriedade têm os valores iguais, quando armazenados num array ordenado?
   Resposta:
   Os valores iguais estão todos juntos, i.e., estão armazenados em posições contíguas do array.

   (b) Que propriedade deverá satisfazer a variável `size` quando o algoritmo terminar?
   Resposta:
   A variável `size` deverá ter o número de ocorrências do valor que se repete mais vezes no array.

   (c) Que região do array tem de ser considerada em cada iteração i do algoritmo, por forma a garantir a actualização correcta da variável `size`?
   Resposta:
   Terão que ser consideradas duas posições contíguas do array. (Por exemplo: A[i] e A[i-1].)

   (d) Que teste tem de ser efectuado para que essa actualização seja correcta?
   Resposta:
   Tem que se testar se os valores que estão nas duas posições contíguas são iguais (e se o contador do número de ocorrências do valor actual é superior ao número de ocorrêcias máximo encontrado até ao momento.)

2. Com base na resposta à alínea anterior, apresente um algoritmo que resolva o problema em questão, completando a função `maiorRepet`. Escreva ainda um invariante de ciclo para o algoritmo que escreveu, e utilize-o para demonstrar a sua correcção.

   Resposta:

Nome:_____ **Número:**_____ **Curso:**_____

```
int maiorRepet(int A[], int n) {
   int i;
   int size;
   int count;

   if (n<=0) return 0;
   else if (n==1) return 1;
   else {
     size=1;
     count=1;

     for (i=1; i<n; i++) {
       if (A[i-1]==A[i]) {
          count=count+1;
          if (count>size) size=count;
       }
       else count=1;
     }
     return size;
   }
}
```

Invariante de ciclo (para o ciclo for).

> *No início de cada iteração* `i`*, a variável* `size` *contém o maior número de valores iguais que ocorre no array* `A` *nas posições* `[0,...,i-1]`*, e a variável* `count` *contém o número de ocorrências do valor que está em* `A[i-1]`*, no array* `A` *nas posições* `[0,...,i-1]`*.*

Ou, de modo mais sintético: No início de cada iteração `i`,

> $\texttt{size}_i$ = o maior número de valores iguais no sub-array `A[0,...,i-1]`
>
> $\texttt{count}_i$ = o número de ocorrências do valor `A[i-1]` no sub-array `A[0,...,i-1]`

Inicialização Antes da primeira iteração, `i=1`, e temos: `size=1` que é maior número de valores iguais no sub-array `A[0]`; e `count=1` que é o número de ocorrências do valor `A[0]` no sub-array `A[0]`. Portanto o invariante verifica-se.

Preservação Assume-se que o invariante é válido no início da iteração `i`, e queremos provar que ele é válido no início da iteração `i+1`. Analizando o algoritmo, podemos afirmar que:

- Se `A[i-1]==A[i]` então $\texttt{count}_{i+1} = \texttt{count}_i + 1$, logo $\texttt{count}_{i+1}$ = o número de ocorrências do valor `A[i]` no sub-array `A[0,...,i]`; e
  - se $\texttt{count}_{i+1} > \texttt{size}_i$, então é porque o número de ocorrências de `A[i]` em `A[0,...,i]` excede o maior número de valores iguais no sub-array `A[0,...,i-1]`, logo $\texttt{size}_{i+1} = \texttt{count}_{i+1}$ faz com que o invariante se verifice no início da iteração `i+1`.

Nome:_____ **Número:**_____ **Curso:**_____

  - se $\texttt{count}_{i+1} \leq \texttt{size}_i$, então é porque o número de ocorrências de `A[i]` em `A[0,...,i]` não excede o maior número de valores iguais no sub-array `A[0,...,i-1]`, logo, como o $\texttt{size}_{i+1}$ será igual a $\texttt{size}_i$, invariante verifica-se no início da iteração `i+1`.
- Se `A[i-1]`$\neq$`A[i]`, então $\texttt{count}_{i+1} = 1$ e $\texttt{size}_{i+1} = \texttt{size}_i$. Como o valor `A[i]` não pode ser igual a nenhum elemento de `A[0,...,i-1]` (uma vez que o array está ordenado), o invariante verifica-se no início da iteração `i+1`.

Terminação No final do ciclo, `i=n`. O que o invariante nos indica é que no final do ciclo a variável `size` tem o maior número de valores iguais que ocorre no array `A` nas posições `[0,...,n-1]` (e a variável `count` tem o número de ocorrências do valor que está em `A[n-1]`, no array `A` nas posições `[0,...,n-1]`).

Portando, uma vez que o valor de `size` é devolvido no final do algoritmo, podemos afirmar que este algoritmo é correcto.

Nome:_____ **Número:**_____ **Curso:**_____

Métodos de Programação II

LESI / LMCC

15 de Junho 2004

Questão 2 [*Estruturas de Dados*] Uma *heap* é uma árvore binária que verifica a seguinte propriedade: o conteúdo de cada nó é menor ou igual que o conteúdo dos seus descendentes (não havendo, no entanto, qualquer relação de ordem entre os conteúdos das duas sub-árvores de um mesmo nó). A função de inserção pode ser escrita como se segue:

```
typedef struct node {
  int num;
  struct node *esq, *dir;
} *Heap;

void swap (Heap h)
{
  Heap aux = h->esq;
  h->esq = h->dir;
  h->dir = aux;
}

Heap insert (int x, Heap h)
{
  if (!h) {
    Heap new = malloc (sizeof(struct node));
    new -> num = x;
    new -> esq = new -> dir = NULL;
    return new;
  }
  if (x < h->num) {
    h->dir = insert(h->num, h->dir);
    swap(h);
    h->num = x;
  }
  else {
    h->dir = insert(x, h->dir);
    swap(h);
  }
  return h;
}
```

Nome:_____ **Número:**_____ **Curso:**_____

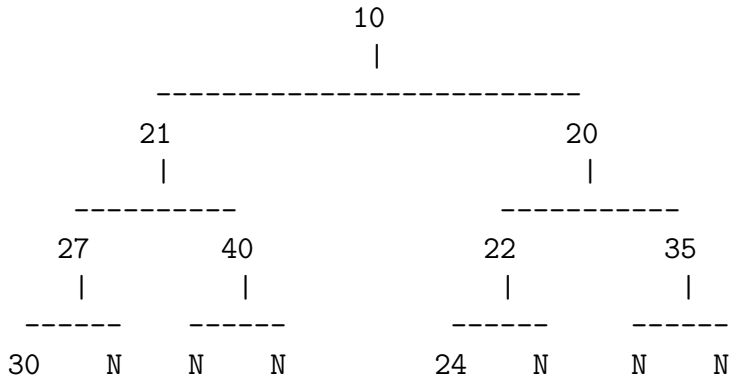
Métodos de Programação II

LESI / LMCC

15 de Junho 2004

1. Desenhe a *heap* que resulta de se inserir consecutivamente os números 10, 20, 30, 22, 21, 35, 40, 24 e 27.

   Resposta:



   Comentário:
   Note-se que em todos os passos recursivos é efectuada uma inserção (recursivamente) na sub-árvore da direita, seguida de uma troca das duas sub-árvores.

2. Com base numa recorrência, efectue a análise assimptótica do tempo de execução da função `insert`.

   Resposta:
   Duas abordagens possíveis: a clássica consiste em escrever a recorrência (N é a dimensão da *heap*)

$$T(N) = \begin{cases} \Theta(1) & \text{se N} = 0 \\ \Theta(1) + T(k) & \text{se } N > 0 \end{cases}$$

   com $0 \leq k < N$. $k = 0$ corresponde ao melhor caso (inserção na *heap* vazia) e $k = N - 1$ ao pior caso. $T(N) = \Omega(1)$ e ao pior caso $T(N) = \mathcal{O}(N)$.

   A *análise agreagda* permite obter limites mais realistas, se nos concentrarmos numa *heap* construída por uma *sequência de N inserções*. É fácil verificar que a função de inserção preserva uma forma balanceada: a sub-árvore da direita tem sempre um número de nós igual ou inferior numa unidade à da esquerda, o que garante uma altura da árvore logarítmica no número de nós. Assim sendo, podemos escrever a recorrência

$$T(N) \approx \begin{cases} \Theta(1) & \text{se N} = 0 \\ \Theta(1) + T(N/2) & \text{se } N > 0 \end{cases}$$

   com solução $T(N) = \Theta(\lg(N))$.

3. Escreva uma função com protótipo

```
int heap2sortedarray(Heap h, int A[], int j);
```

Nome:_____ **Número:**_____ **Curso:**_____

que converte uma *heap* num vector ordenado, começando na posição `j` do vector, e devolve o número de elementos colocados no vector.

Sugestão: Pode utilizar uma de duas estratégias alternativas para tratar os elementos armazenados nas duas *sub-heaps*:

- converter recursivamente cada *sub-heap* numa sequência ordenada, e fundir os resultados; ou
- fundir as duas *sub-heaps* numa só estrutura do mesmo tipo, e prosseguir recursivamente.

Resposta:
Seguindo a primeira sugestão:

```
int heap2sortedarray(Heap h, int A[], int j)
{
  int a, b;
  if (!h) return 0;
  A[j] = h->num;
  a = heap2sortedarray(h->esq, A, j+1);
  b = heap2sortedarray(h->dir, A, j+a+1);
  merge(A, j+1, j+a, j+a+b);
  return 1+a+b;
}
```

Comentário:
`merge` **é uma função (usada por exemplo no algoritmo mergesort) que funde dois vectores ordenados (ou melhor, duas sub-sequências contíguas de um mesmo vector, o que evita a utilização de espaço extra). Note-se que a segunda invocação recursiva vai preencher o vector começando na primeira posição ainda não utilizada.**

4. Com o auxílio de uma *heap* é possível ordenar uma sequência como se segue:

```
void hsort (int A[], int n)
{
  int i;
  Heap h = NULL;
  for (i=1; i<=n; i++)
    h = insert(A[i], h);
  heap2sortedarray(h, A, 1);
}
```

Analise o tempo de execução deste algoritmo.

Resposta:

Nome:_____ **Número:**_____ **Curso:**_____

Começamos por analisar `heap2sortedarray`:

$$T_{heap2sortedarray}(N) \approx \begin{cases} \Theta(1) & \text{se N} = 0 \\ \Theta(N-1) + T(N/2) + T(N/2) & \text{se } N > 0 \end{cases}$$

A respectiva árvore de recursão tem $\lg N + 1$ níveis, sendo o custo total das operações `merge` em cada nível $\mathcal{O}(N)$. Temos então $T_{heap2sortedarray}(N) = \mathcal{O}(N \lg N)$ e para `hsort`:

$$T(N) = \sum_{i=0}^{N-1} T_{insert}(i) + T_{heap2sortedarray}(N) = \sum_{i=0}^{N-1} \Theta(\lg i) + \mathcal{O}(N \lg N) = \mathcal{O}(N \lg N)$$

Métodos de Programação II

LESI / LMCC

15 de Junho 2004

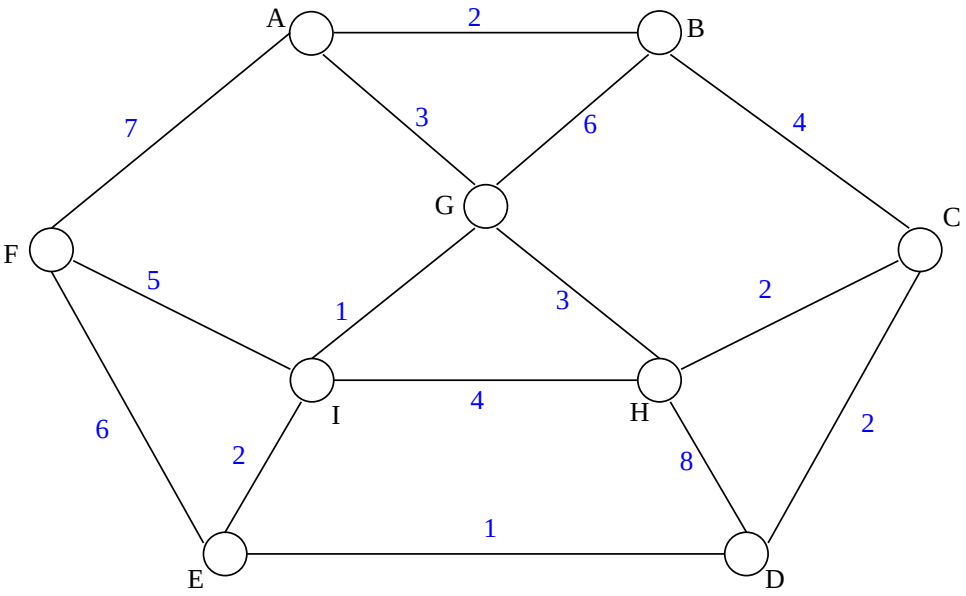
Questão 3 [*Algoritmos de Grafos*] A versão simétrica do algoritmo de Dijkstra para o cálculo do caminho mais curto (entre um par de nós A e B, num grafo não-orientado) consiste na execução alternada de um passo do algoritmo a partir de cada nó, o que implica construir simultaneamente duas árvores (V_A, T_A) e (V_B, T_B), com raízes A e B. Esta versão permite optimizar a execução do algoritmo em certos casos.

1. Durante a execução do algoritmo simétrico, caso exista um caminho entre X e Y, atingir-se-á um estado em que um nó pertence a ambas as árvores V_X e V_Y. É então garantido que o caminho mais curto entre X e Y só pode passar por nós contidos em $V_X \cup V_Y$. Diga justificando se esta afirmação é verdadeira ou falsa.

   Resposta:
   A afirmação é verdadeira. O algoritmo acrescenta arcos às duas árvores por forma a que os nós que vão sendo incluídos estão progressivamente mais distantes das origens X e Y. Sendo assim, quando as árvores se intersectam num nó Z, qualquer nó W não pertencente a V_X nem a V_Y é necessariamente mais distante de X e de Y do que Z (que pertence a ambas as árvores), pelo que qualquer caminho entre X e Y passando por W é mais longo do que o caminho que passa por Z. Prova-se assim que o caminho mais curto entre X e Y não pode passar por nós não contidos numa das árvores (ou seja, em $V_X \cup V_Y$), como se pretendia.

   Comentário:
   Note-se que isto não garante que o caminho mais curto passe por Z; pode haver caminhos mais curtos do que o que passa por Z, passando apenas por outros nós das árvores. De notar ainda que uma formulação mais precisa afirmaria que existe um caminho mais curto entre X e Y que só passa por nós contidos em $V_X \cup V_Y$ (tendo em conta que o caminho mais curto não é necessariamente único).

2. Aplique o algoritmo simétrico para calcular o caminho de F para C no grafo seguinte. Diga justificando se este algoritmo optimizado apresenta vantagens em relação ao algoritmo tradicional, neste caso concreto.



Nome:_____ **Número:**_____ **Curso:**_____

Resposta:

Algoritmo simétrico – sequência de arcos acrescentados às árvores V_F e V_C:

$$\begin{array}{ll}(F,I) & (C,H)\\ (F,E) & (C,D)\\ (I,G) & (D,E)\end{array}$$

Algoritmo pára; árvores intersectam-se em E.

Algoritmo directo – sequência de arcos acrescentados à árvore V_F:

$$\begin{array}{c}(F,I)\\ (F,E)\\ (I,G)\\ (F,A)\\ (E,D)\\ (D,C)\end{array}$$

Usando como medida do trabalho realizado o número de arcos (ou nós) acrescentados às árvores, verifica-se que as execuções dos dois algoritmos não apresentam diferenças significativas.

Comentário:
Pretendia-se aqui apenas que se apresentasse a execução de ambos os algoritmos e se definisse uma medida razoável de trabalho efectuado. Outras execuções eram possíveis; por exemplo no algoritmo directo, dada a escolha entre arcos equidistantes da origem, poder-se-ia acrescentar antes de (D,C) os arcos (A,B) e (I,H).

Nome:_____ **Número:**_____ **Curso:**_____

Métodos de Programação II

LESI / LMCC

15 de Junho 2004

Questão 4 [*Problemas NP-Completos*]
Considere o seguinte problema:

> **Problema**: Cobertura Exacta por Conjuntos de 3 Elementos
> **Input**: Um conjunto X, de tamanho múltiplo de 3, e uma colecção C, de sub-conjuntos de 3 elementos de X.
> **Pergunta**: Será que C contém uma cobertura exacta de X, isto é, existe uma sub-colecção C' de C, tal que cada elemento de X apareça uma e uma só vez em C'?
> **Exemplo**:
> X = {1,2,3,4,5,6,7,8,9},
> C={{1,2,3},{2,3,4},{4,5,6},{7,8,9}},
> C'={{1,2,3},{4,5,6},{7,8,9}}.

1. Explique o conceito de problema de decisão NP-completo, e diga que formas conhece de demonstrar que um problema é NP-completo, com base na noção de redução polinomial.

   Resposta:
   A classe de problemas NP é constituída por todos os problemas de decisão que são resolúveis através de um algoritmo não determinístico limitado polinomialmente.

   A redução polinomial de um problema Π_1 a um problema Π_2 da classe NP, é o processo de encontrar uma função de transformação, que execute em tempo polinomial, dos *inputs* de Π_1 para os de Π_2 tal que:

   - Dado um qualquer algoritmo não determinístico limitado polinomialmente que resolva Π_2,
   - e um qualquer *input* x para o problema Π_1,
   - a resposta de Π_1 a x seja "Sim" se e só se a resposta de Π_2 à transformação de x seja "Sim".

   Os problemas NP-completos constituem uma sub-classe da classe NP. Um problema Π diz-se NP-completo, se for possível demonstrar que todos os problemas em NP são redutíveis polinomialmente ao problema Π.

   Assim, uma das formas de demonstrar que um problema Π_1 é NP-completo consiste em, de acordo com a definição, provar que todos os problemas na classe NP são redutíveis polinomialmente a Π_1.

   Uma segunda forma de fazer esta demonstração consiste em provar que um problema Π_2, para o qual já tenha sido demonstrado ser NP-completo, é redutível polinomialmente a Π_1. Esta aproximação é fundamentada pela transitividade da operação de redução polinomial:

   - se Π_2 é NP-completo, então todos os problemas em NP são redutíveis polinomialmente a Π_2;
   - se Π_2 é redutível polinomialmente a Π_1, então todos os problemas em NP são também redutíveis polinomialmente a Π_1,
   - donde se pode concluir que Π_1 é também NP-completo.

   Comentário:
   Esta resposta está "demasiado completa": todos os conceitos envolvidos estão explicados por motivos pedagógicos.

Nome:_____ **Número:**_____ **Curso:**_____

2. Sabendo que o problema apresentado acima é NP-completo, demonstre que o seguinte problema é também NP-completo:

   Problema: Cobertura Mínima
   Input: Um conjunto X e uma colecção C, de sub-conjuntos de X. Um inteiro K.
   Pergunta: Será que C contém uma cobertura de X de tamanho K, isto é, existe uma sub-colecção C' de C, com K ou menos sub-conjuntos de X, tal que cada elemento de X apareça uma e uma só vez em C'?

Resposta:
Vamos utilizar o segundo método descrito na alínea anterior para efectuar esta demonstração.
Para isso temos de demonstrar que o *Problema da Cobertura Exacta por Conjuntos de 3 Elementos*, que é NP-completo, é redutível polinomialmente ao *Problema da Cobertura Mínima*.
Esta demonstração é imediata constatando que o primeiro problema é um sub-problema do segundo.
De facto, usando como transformação polinomial de um *input* (X, C) para o primeiro problema, num *input* $(X, C, K = |X|/3)$ para o segundo problema, temos a garantia que:

- se a reposta correcta ao *input* (X, C) no *Problema da Cobertura Exacta por Conjuntos de 3 Elementos* for "Sim" então, garantidamente, um algoritmo que resolva correctamente o *Problema da Cobertura Mínima* devolverá a resposta "Sim" ao *input* $(X, C, K = |X|/3)$;

- por outro lado, para todos os *inputs* (X, C, K) do *Problema da Cobertura Mínima* em que (X, C) formem um *input* válido para o *Problema da Cobertura Exacta por Conjuntos de 3 Elementos* e $K \geq |X|/3$, uma resposta "Sim" devolvida por um algoritmo que resolva o *Problema da Cobertura Mínima* implica que a resposta correcta, nesse caso, para o *Problema da Cobertura Exacta por Conjuntos de 3 Elementos* é também "Sim".

Nome:_____ **Número:**_____ **Curso:**_____