Ficha 2

Análise da complexidade de programas

Algoritmos e Complexidade
LEI / LCC / LEF

1 Contagem

1. Para cada uma das funções de ordenação abaixo

   - Identifique o melhor e pior casos em termos do número de comparações entre elementos do array e em termos do número de trocas efectuadas.
   - Calcule o número de comparações entre elementos do array efectuadas nesses casos identificados.

   (a)
   ```
   void bubbleSort (int v[], int N){
         int i, j;
         for (i=N-1; i>0; i--)
             for (j=0; j<i; j++)
                 if (v[j] > v[j+1]) swap (v,j,j+1);
   }
   ```

   (b)
   ```
   void iSort (int v[], int N){
         int i, j;
         for (i=1; i<N; i++)
             for (j=i; j>0 && v[j-1] > v[j]; j--)
                 swap (v,j,j-1);
   }
   ```

2. Considere as seguintes definições de funções (já estudadas na Ficha 1) que calculam o produto de dois números inteiros não negativos.

   ```
   int mult1 (int x, int y){
      // pre: x>=0
      int a=x, b=y, r=0;
      while (a>0){
         r = r+b; a = a-1;
      }
      // pos: r == x * y
      return r;
   }
   ```

   ```
   int mult2 (int x, int y){
      // pre: x>=0
      int a=x, b=y, r=0;
      while (a>0) {
         if (a%2 == 1) r = r+b;
         a=a/2; b=b*2;
      }
      // pos: r == x * y
      return r;
   }
   ```

Para cada uma destas funções efectue uma contagem do número de vezes que as operações primitivas[1] (`+ - *2 /2 %2`) contidas no corpo do ciclo são executadas no pior caso.

Considere que o tamanho do input é *o número de* bits *necessários para representar os números inteiros passados como argumento.* Recorde que, por exemplo, os números cuja representação requer 5 bits são $\{16, \ldots, 31\}$.

3. Considere a seguinte definição de uma função que calcula a maior soma de um segmento de um array de inteiros.

```
int maxSoma (int v[], int N) {
  int i, j, r=0, m;
  for (i=0; i<N; i++)
    for (j=i; j<N; j++) {
      m = soma(v,i,j);
      if (m>r) r = m;
    }
  return r;
}
```

```
int soma (int v[], int a, int b) {
  int r = 0, i;
  for (i=a; i<=b; i++)
    r=r+v[i];
  return r;
}
```

   (a) Determine a complexidade da função `maxSoma` em termos do número de acessos ao array argumento.

   (b) Uma forma alternativa de resolver este problema consiste em usar um array auxiliar `c` com `N` elementos, que será preenchido de acordo com a seguinte propriedade

   o elemento `c[i]` contem a maior soma de um segmento do array que termina (e inclui) `v[i]`.

   Implemente esta estratégia e compare a complexidade desta solução com a da função apresentada.

4. Considere a seguinte função que calcula o comprimento do maior segmento crescente de uma array de inteiros.

```
int crescente (int v[], int N) {
  int i;
  for (i=1; i<N; i++)
    if (v[i] < v[i-1]) break;
  return i;
}
```

```
int maxcresc (int v[], int N) {
  int r = 1, i = 0, m;
  while (i<N-1) {
    m = crescente (v+i, N-i);
    if (m>r) r = m;
    i++;
  }
  return r;
}
```

Identifique o melhor e pior caso da função `maxcresc` em termos do número de comparações entre elementos do array argumento.

[1] Note que as operações `*2`, `/2` e `%2` se podem escrever como `>>1`, `<<1` e `&1`

Calcule ainda esse número para o pior caso identificado.

Uma possível optimização desta função consiste em substituir o incremento `i++` do corpo do ciclo por `i+=m`. Qual o número de comparações efectuadas por esta alternativa no pior caso identificado acima?

2 Definições Recursivas

1. Utilize uma árvore de recorrência para encontrar limites superiores para o tempo de execução dados pelas seguintes recorrências (assuma que para todas elas $T(0)$ é uma constante):

   (a) $T(n) = k + T(n-1)$ com k constante

   (b) $T(n) = k + T(n/2)$ com k constante

   (c) $T(n) = k + 2 * T(n/2)$ com k constante

   (d) $T(n) = n + T(n-1)$

   (e) $T(n) = n + T(n/2)$

   (f) $T(n) = n + 2 * T(n/2)$

2. Exprima a complexidade da função `maxSomaR` (em termos do número de acessos ao array argumento) como uma recorrência.

```
int maxSomaR (int v[], int N) {
  int r=0, m1, m2, i;
  if (N>0) {
    m1 = m2 = v[0];
    for (i=1; i<N; i++) {
      m2 = m2+v[i];
      if (m2>m1) m1=m2;
    }
    m2 = maxSomaR (v+1,N-1);
    if (m1>m2) r = m1; else r = m2;
  }
  return r;
}
```

3. Considere o seguinte algoritmo para o problema das *Torres de Hanoi*:

```
void Hanoi(int nDiscos, int esquerda, int direita, int meio)
{
  if (nDiscos > 0) {
    Hanoi(nDiscos-1, esquerda, meio, direita);
    printf("mover disco de %d para %d\n", esquerda, direita);
    Hanoi(nDiscos-1, meio, direita, esquerda);
  }
}
```

Escreva uma relação de recorrência que exprima a complexidade deste algoritmo (por exemplo, em função do número de linhas impressas). Desenhe a árvore de recursão do algoritmo e obtenha a partir dessa árvore um resultado sobre a sua complexidade assimptótica.

4. Considere a seguinte definição da função que ordena um vector usando o algoritmo de *merge sort*.

```
void msort (int v[], int N) {
   int m = N/2;
   if (N>1) {
      msort (v, m); msort (v+m, N-m);
      mergeH (v, N);
   }
}
```

Considere que a função `int mergeH (int a[], int N)` executa em tempo $T_{\text{merge}}(N) = 2 * N$. Apresente uma relação de recorrência que traduza o tempo de execução de `msort` em função do tamanho do vector argumento.

Apresente ainda uma solução dessa recorrência.

5. Considere a definição da função `altura` sobre árvores binárias Descreva a sua complexidade com uma recorrência, considerando duas configurações extremas de árvores: (1) árvores equilibradas (os elementos estão distribuidos uniformemente pelas duas sub-árvores) ou (2) árvores "lista" (em que cada nodo tem pelo menos uma das sub-árvores vazias).

```
int altura (ABin a){
    int r=0;
    if (a!=NULL)
       r = 1 + max (altura (a->esq),
                    altura (a->dir));
    return r;
}
```

3 Análise de caso médio

1. Relembre a função `crescente` definida acima.

```
int crescente (int v[], int N) {
  int i;
  for (i=1; i<N; i++)
    if (v[i] < v[i-1]) break;
  return i;
}
```

Considerando que os valores do array são perfeitamente aleatórios e por isso, para qualquer índice i, a probabilidade de a posição `i` conter um valor menor do que a posição `i-1` é 0.5.

Calcule o número médio de comparações efectuadas por esta função.

Com base no resultado obtido, calcule o custo médio da função `maxcresc` apresentada na Secção 1.

2. Com as mesmas considerações feitas na alínea anterior, num array aleatório `v`, e para cada posição `i`, em média, metade dos elementos de `v` entre as posições `0` e `i` são maiores do que `v[i]`.

   Com base neste facto, faça a análise de tempo médio da função `iSort` apresentada na Secção 1.

3. Considere a seguinte variante da função `strncmp` que determina o primeiro índice em que duas strings diferem.

```
int strNdif (char s1[], char s2[], int N){
    int i;
    for (i=0; i<N && s1[i] && s1[i] == s2[i]; i++)
        ;
    return i;
}
```

   Determine o custo médio (número de comparações entre elementos dos arrays) desta função, quando invocada apenas com strings de letras minúsculas (assuma por isso que a probabilidade de duas letras serem iguais é de $\frac{1}{26}$)

4. Considere a seguinte definição da função `inc` que recebe um array de `N` bits (representando um inteiro `x`) e que modifica o array de forma a representar `x+1`.

```
int inc (int x[], int N){
    int i=N-1;
    while (i>=0 && x[i] == 1)
         x[i--] = 0;
    if (i<0) return 1;
    x[i] = 1;
    return 0;
}
```

   Calcule o custo médio desta função (em termos do número de bit alterados).

5. Relembre o algoritmo de procura numa árvore binária de procura.

```
int elem (int x, ABin a){
    while (a != NULL && a->valor != x)
```

```
        if (x < a->valor) a = a->esq;
        else a = a->dir;
    return (a!=NULL);
}
```

Analise a complexidade média (em termos de número de nodos consultados) desta função, assumindo que o elemento que se procura existe com igual probabilidade em cada posição da árvore.

Faça esta análise para duas configurações extremas de árvores: (1) árvores equilibradas (os elementos estão distribuidos uniformemente pelas duas sub-árvores) ou (2) árvores "lista" (em que cada nodo tem pelo menos uma das sub-árvores vazias).

6. Faça a análise de caso médio da função `f`, assumindo que `a` é um array de 0s e 1s e que `g` é uma função da classe $\Theta(2^N)$. Comece por identificar o melhor e o pior casos.

```
int f(int a[], int N) {
   int b = 0;
   for (int i = 0; i < N; i++)
       if (a[i] == 1) b++;
   if (b == 1) return g(a,N);
   else return 0;
}
```

4 Análise amortizada

1. Uma implementação possível de uma fila de espera (*Queue*) utiliza duas *stacks* **A** e **B**, por exemplo:

```
typedef struct queue {
 Stack a;
 Stack b;
} Queue;
```

- A inserção (`enqueue`) de elementos é sempre realizada na *stack* **A**;
- para a remoção (`dequeue`), se a *stack* **B** não estiver vazia, é efectuado um *pop* nessa *stack*; caso contrário, para todos os elementos de **A**, faz-se sucessivamente *pop* e *push* na *stack* **B**. Faz-se depois *pop* na stack **B**, que é devolvido como resultado.

(a) Efectue a análise do tempo de execução no melhor e no pior caso das funções `enqueue` e `dequeue`, assumindo que todas as operações das *stacks* são realizadas em tempo constante.

(b) Mostre que o custo amortizado de cada uma das operações de `enqueue` ou `dequeue` numa sequência de N operações é $\mathcal{O}(1)$ usando o método do potencial.

2. Considere-se uma estrutura de dados do tipo stack com a habitual operação 'push', mas em que a operação 'pop' é substituída por uma operação 'multipop', uma generalização que remove os k primeiros elementos, deixando a pilha vazia caso contenha menos de k elementos. Uma implementação possível será

```
void multiPop(S,k) {
      while (!IsEmpty(S) && k != 0) {
        pop(S);
        k -= 1;
      }
    }
```

   Pela análise tradicional de pior caso,

   - 'push' executa em tempo $\mathcal{O}(1)$
   - 'multiPop' executa em tempo $\mathcal{O}(N)$

   Utilize um dos métodos estudados, para mostrar que em termos amortizados a operação 'multiPop' executa também ela em tempo constante $\mathcal{O}(1)$.

3. Considere um algoritmo de inserção ordenada numa lista (crescente), com uma particularidade: são apagados os nós iniciais da lista contendo valores inferiores ao que está a ser inserido. Por exemplo, a inserção de 30 na lista [10, 20, 40, 50] resulta na lista [30, 40, 50]. A função seguinte implementa este algoritmo em C.

```
node *insert_rem (node *p, int x) {
  node *new = malloc(sizeof(node)); new->value = x;
  while (p && x > p->value)
    { aux = p; p = p->next; free (aux); }
  new->next = p;
  return new;
}
```

   (a) Analise o tempo de execução assimptótico de 'insert_rem', identificando o pior e o melhor caso.

   (b) Em termos amortizados a operação de inserção da questão anterior executa em tempo constante. Efectue a sua análise agregada considerando a sequência de inserções 20, 70, 60, 30, 40, 50, 10, 80 (partindo de uma lista vazia). Considere que o custo real de cada inserção/remoção efectuada à cabeça da lista é 1, por isso a inserção de 30 na lista [10, 20, 40, 50] tem custo 3. Apresente ainda uma função de potencial apropriada sobre as listas, e calcule a partir dela o custo amortizado constante desta operação 'insert_rem'.