Ficha 2

Algoritmos e Complexidade

Contagem de operações em algoritmos iterativos

1 Procura em vectores

1. Considere a seguinte função que procura um valor num *array*.

```
int xlsearch (int a[], int N, int x) {
int i;

i=0;
while (i<N) && (a[i] != x)
    i++;
if (i<N) return i;
else return (-1);
}
```

   (a) Identifique o melhor e o pior caso de execução desta função.

   (b) Para cada um dos casos identificados acima, apresente o tempo de execução desta função, em função do tamanho do *array* de entrada.

2. Considere agora uma outra versão desta mesma função que assume que o *array* de entrada está ordenado por ordem crescente.

```
int lsearch (int a[], int N, int x) {
int i;

i=0;
while (i<N) && (a[i] > x)
    i++;
if (i<N) && (a[i] == x) return i;
else return (-1);
}
```

   (a) Identifique o melhor e o pior caso de execução desta função.

   (b) Para cada um dos casos identificados acima, apresente o tempo de execução desta função, em função do tamanho do *array* de entrada.

   (c) Apresente o tempo médio de execução desta função, em função do tamanho do *array* de entrada. Para isso assuma que a probabilidade de o ciclo fazer um qualquer número de iterações é constante.

(d) Repita este cálculo para a função da alínea anterior. Para isso, assuma que os inteiros em causa têm W *bits* (existindo por isso 2^W inteiros diferentes). A probabilidade de encontrar o número numa dada posição é por isso de $\frac{1}{2^W}$

3. Considere uma última versão desta função que vai reduzindo a gama de valores a procurar.

```
int bsearch (int a[], int N, int x) {
int l, u, m;

l=0; u=N-1;
while l<u {
    m = (l+u)/2;
    if (a [m] == x) l = u = m;
    else if (a[m] > x) u = m-1;
    else l = m+1;
}
if (a[l] == x) return l;
else return (-1);
}
```

(a) Identifique o melhor e o pior caso de execução desta função.

(b) Apresente o tempo de execução desta função, em função do tamanho do *array* de entrada, para o pior caso identificado acima.

2 Ordenação

Nesta secção vamos analisar algumas funções de ordenação de vectores. Para cada um dos algoritmos apresentados,

- Descreva (informalmente) o invariante do ciclo mais exterior.
- Identifique o melhor e o pior casos de execução dessas funções.
- Para cada um dos casos identificados na ponto anterior determine
  1. o número de comparações entre elementos do *array*.
  2. o número de trocas (chamadas à função `swap`).
- Considere um caso adicional que corresponde a um *array* parcialmente ordenado em que o único elemento *fora de ordem* é o que está na primeira posição e que corresponde ao maior elemento do *array*.

Troca directa

```
void swapSort (int v[], int N) {
int i, j;
```

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

Min-sort

```
void minSort (int v[], int N) {
int i, j, m;

for (i=0; (i<N-1); i++) {
    m = i;
    for (j=i+1; (j<N); j++)
        if (v [m] > v [j]) m = j;
    if (i != m) swap (v,i,m);
}
}
```

Bubble-sort

```
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);
}
```

Bubble-sort opt

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

3 Operações aritméticas

1. Considere os seguintes definições de funções que calculam o produto de dois números inteiros não negativos.

```
int prod (int x, int y) {
int r;

r = 0;
while x>0 {
    r = r+y;
    x = x-1;
   }
return r;
}
```

```
int bprod (int x, int y) {
int r;

r = 0;
while x>0 {
    if (x % 2 == 1) r = r + y;
    y = y * 2;
    x = x / 2;
   }
return r;
}
```

(a) Para cada uma das soluções apresentadas, identifique o melhor e pior casos em termos do tempo de execução destas funções. Faça a sua análise, em função do número de *bits* usados para representar os números inteiros em causa.

(b) Determine o tempo de execução de cada uma das soluções no pior caso, em função do número de *bits* usado para representar inteiros.

2. Considere a seguinte definição de uma função que calcula a potência inteira de um número.

```
float pot (float b, int e) {
float r;

r = 1;
while e>0 {
    r = r * b;
    e = e - 1;
   }
return r;
}
```

Apresente uma versão alternativa desta função cujo número de multiplicações, no pior caso, seja proporcional ao número de *bits* usados para representar o expoente (Sugestão: use como inspiração as funções apresentadas na questão anterior para calcular o produto de dois números).

3. Considere o seguinte algoritmo para o problema do cálculo do valor de um polinómio

$$P(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$$

num ponto x dado, sendo o polinómio representado por um vector de coeficientes (ordenado por ordem crescente do grau):

```
float Poly (float a[], int n, float x) {
float p, xpotencia;
int i;
```

```
p = a[0] + a[1] * x;
xpotencia = x;
for (i=2 ; i<=n ; i++) {
    xpotencia = xpotencia * x;
    p = p + a[i] * xpotencia;
}
return p;
}
```

(a) Quantas operações de soma e multiplicação efectua este algoritmo?

(b) O *algoritmo de Horner* é uma alternativa mais eficiente para a resolução do mesmo problema. Trata-se de uma optimização do algoritmo anterior, que efectua uma *factorização* do polinómio (note-se que $ab+ac$ pode ser calculado com apenas uma multiplicação como $a(b+c)$):

```
float HornerPoly (float a[], int n, float x) {
float p;
int i;

p = a[n];
for (i=n-1 ; i>=0 ; i--)
    p = p*x + a[i];
return p;
}
```

Quantas operações de soma e multiplicação efectua este algoritmo?

4 Outros algoritmos iterativos

1. Considere a seguinte função em C que determina se um vector de inteiros contem elementos repetidos.

```
int repetidos (int v[], int N) {
int rep = 0;
int i, j;
for (i=0; (i<N-1) && !rep; i++)
    for (j=i+1; (j<N) && !rep; j++)
        if v[i] == v[j] rep = 1;
return rep;
}
```

   (a) Identifique o melhor e o pior casos da execução desta função.

   (b) Para o pior caso definido acima, calcule o número de comparações (entre elementos do vector) que são efectuadas (em função do tamanho do *array* argumento).

2. Considere a seguinte função em C que calcula o número de elementos diferentes de um *array* de inteiros.

```
int diferentes (int v[], int N) {
int dif = 0;
int i, j;
for (i=0; (i<N); i++){
    for (j=i+1; (j<N) && (v[i] != v[j]); j++);
    if (j==N) dif++
return dif;
}
```

(a) Identifique o melhor e o pior casos da execução desta função.

(b) Para o pior caso definido acima, calcule o número de comparações (entre elementos do vector) que são efectuadas (em função do tamanho do *array* argumento).

3. A função abaixo recebe como argumento um *array* de *bits* que representa um número inteiro (em que o *bit* menos significativo está na posição 0) e incrementa esse número.

```
void inc (int b[], int N) {
int i;

i = 0;
while (i < N) && (b[i] == 1){
    b[i] = 0;
    i++;
   }
if (i<N) b[i] = 1;
}
```

(a) Identifique o melhor e o pior caso de execução desta função.

(b) Para cada um dos casos identificados acima, apresente o tempo de execução desta função, em função do tamanho do *array* de entrada.

(c) Apresente o tempo médio de execução desta função, em função do tamanho do *array* de entrada. Para isso assuma que o número armazenado no *array* é aleatório, i.e., e probabilidade de uma dada posição do *array* ser 0 ou 1 é de 1/2.

5 Análise Assimptótica

1. Sejam f e g duas funções (entre números reais).

Considere então as seguintes definições de relações entre funções:

$$
\begin{array}{rcl}
f(x) \sim g(x) & \text{sse} & \lim_{x\to\infty} \frac{f(x)}{g(x)} = 1 \\
f(x) \in o(g(x)) & \text{sse} & \lim_{x\to\infty} \frac{f(x)}{g(x)} = 0 \\
f(x) \in \mathcal{O}(g(x)) & \text{sse} & \exists_{C>0} \exists_{N_0} \forall_{n>N_0}\, f(n) \leq C * g(n) \\
f(x) \in \Theta(g(x)) & \text{sse} & f(x) \in \mathcal{O}(g(x)) \;\text{ e }\; g(x) \in \mathcal{O}(f(x))
\end{array}
$$

(a) Mostre que todas estas (4) relações são transitivas.

(b) Mostre que tanto Θ como $\mathcal{O}$ distribuem relativamente à soma, i.e., que

- se $f(x) \in \mathcal{O}(a(x))$ e $g(x) \in \mathcal{O}(b(x))$ então $(f(x) + g(x)) \in \mathcal{O}(a(x) + b(x))$
- se $f(x) \in \Theta(a(x))$ e $g(x) \in \Theta(b(x))$ então $(f(x) + g(x)) \in \Theta(a(x) + b(x))$

2. Diga justificando quais das seguintes igualdades são verdadeiras:

(a) $(x^2 + 3 * x + 1)^3 \sim x^6$

(b) $\frac{(\sqrt{x}+1)^3}{(x^2+1)} = o(1)$

(c) $e^{\frac{1}{x}} = \Theta(1)$

(d) $\frac{1}{x} \sim 0$

(e) $x^3 * (\log \log x)^2 = o(x^3 * \log x)$

(f) $\log x + 1 = \mathcal{O}(\log \log x)$

(g) $\sin x = \mathcal{O}(1)$

(h) $\sin x = o(1)$

(i) $\frac{\cos x}{x} = o(1)$

(j) $\sum_{j=0}^{j \leq x} 1 = \mathcal{O}(x)$

A Somas de séries

1. $\sum_{i=a}^{b} 1 = b - a + 1$ (corresponde ao número de parcelas do somatório em causa).
2. $\sum_{i=a}^{b} i = \frac{(a+b)(b-a+1)}{2}$.
   Uma forma de obter esta igualdade (usando a igualdade anterior) é calcular a expressão $\sum_{i=a}^{b}(i+1)^2$ de duas formas diferentes:
   - Por um lado temos que
   $$\begin{array}{rcl} \sum_{i=a}^{b}(i+1)^2 & = & \sum_{i=a}^{b}(i^2 + 2 * i + 1) \\ & = & (\sum_{i=a}^{b} i^2) + 2 * (\sum_{i=a}^{b} i) + (\sum_{i=a}^{b} 1) \end{array}$$
   - Por outro lado,
   $$\begin{array}{rcl} \sum_{i=a}^{b}(i+1)^2 & = & (a+1)^2 + (a+2)^2 + \cdots + b^2 + (b+1)^2 \\ & = & -(a^2) + a^2 + (a+1)^2 + (a+2)^2 + \cdots + b^2 + (b+1)^2 \\ & = & -(a^2) + (\sum_{i=a}^{b} i^2) + (b+1)^2 \\ & = & (\sum_{i=a}^{b} i^2) + (b+1)^2 - (a^2) \end{array}$$

   igualando estes termos temos que
   $$(\sum_{i=a}^{b} i^2) + 2 * (\sum_{i=a}^{b} i) + (\sum_{i=a}^{b} 1) = (\sum_{i=a}^{b} i^2) + (b+1)^2 - (a^2)$$
   Daí que,
   $$\sum_{i=a}^{b} i = \frac{(b+1)^2 - (a^2) - (\sum_{i=a}^{b} 1)}{2}$$
3. Use a estratégia apresentada acima para derivar a igualdade
   $$\sum_{i=1}^{n} i^2 = \frac{n * (n+1) * (2 * n + 1)}{6}$$
4. $\sum_{i=1}^{n} b^i = b * \frac{b^n - 1}{b - 1}$
   Uma forma de obter esta igualdade é:
   - Expandido o somatório
   $$S = \sum_{i=1}^{n} b^i = b^1 + b^2 + b^3 + \cdots + b^{n-1} + b^n$$
   - Multiplicando ambos os membros desta igualdade por b,
   $$\begin{array}{rcl} S * b & = & b^2 + b^3 + \ldots + b^n + b^{n+1} \\ & = & (-b + b) + b^2 + b^3 + \ldots + b^n + b^{n+1} \\ & = & -b + (\sum_{i=1}^{n} b^i) + b^{n+1} \\ & = & S + b * (b^n - 1) \end{array}$$

   E daí que
   $$\begin{array}{rcl} S * b - S & = & b * (b^n - 1) \\ S * (b - 1) & = & b * (b^n - 1) \end{array}$$

5. Derive, a partir da anterior, uma igualdade para calcular $\sum_{i=a}^{n} b^i$

6. Um caso particular da igualdade anterior (para $b = 2$) tem uma derivação mais intuitiva para informáticos. A soma

$$S = \sum_{i=0}^{n} 2^i$$

é representada em base 2 como uma sequência de $(n+1)$ bits a 1. Se lhe somarmos 1, a sua representação binária passa a ser um 1 seguido de $(n+1)$ 0's, i.e., o número 2^{n+1}. Daí que $S + 1 = 2^{n+1}$ e por isso,

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

7. Um outro caso partivular é quando b é $\frac{1}{2}$, i.e., quando queremos calcular

$$F(N) = \sum_{i=1}^{N} \frac{1}{2^i}$$

Para obtermos este resultado de uma outra forma (que não a apresentada acima), atentemos na seguinte série de somas:

$$\begin{array}{l}
1 \\
\frac{1}{2} + \frac{1}{2} \\
\frac{1}{2} + \frac{1}{4} + \frac{1}{4} \\
\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{8} \\
\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{16} \\
\cdots \\
\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^N} + \frac{1}{2^N}
\end{array}$$

De facto todas estas somas têm o mesmo valor (de 1): cada soma resulta da anterior substituindo a última parcela por uma soma com o mesmo valor.

Por outro lado, a última das somas apresentadas pode ser escrita como $F(N) + \frac{1}{2^N}$. Pelo que podemos concluir que

$$F(N) = \sum_{i=1}^{N} \frac{1}{2^i} = 1 - \frac{1}{2^N}$$

8. Uma outra soma envolvendo potências de 2 é

$$X(N) = \sum_{i=1}^{N} \frac{i}{2^i} = \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \cdots + \frac{N}{2^N}$$

Um meio de obter uma fórmula par esta soma consiste em organizar as parcelas da seguinte forma:

$$\begin{array}{rclllll}
X(N) & = & \frac{1}{2} & +\frac{2}{4} & +\frac{3}{8} & +\cdots & +\frac{N}{2^N} \\
 & = & \frac{1}{2} & +\frac{1}{4} & +\frac{1}{8} & +\cdots & +\frac{1}{2^N} \\
 & & & +\frac{1}{4} & +\frac{1}{8} & +\cdots & +\frac{1}{2^N} \\
 & & & & +\frac{1}{8} & +\cdots & +\frac{1}{2^N} \\
 & & & & & +\cdots & +\frac{1}{2^N} \\
 & & & & & & +\frac{1}{2^N}
\end{array}$$

$$
\begin{aligned}
X(N) &= F(N)+ \\
&\quad F(N)-F(1)+ \\
&\quad F(N)-F(2)+ \\
&\quad +\cdots+ \\
&\quad F(N)-F(N-1)
\end{aligned}
$$

isto é,

$$
\begin{aligned}
X(N) &= N*F(N)-\textstyle\sum_{i=1}^{N-1}F(i) \\
&= N*F(N)-\textstyle\sum_{i=1}^{N-1}(1-\frac{1}{2^i}) \\
&= N*F(N)-(N-1)+F(N-1)
\end{aligned}
$$

$$X(N)=\sum_{i=1}^{N}\frac{i}{2^i}=2-\frac{N+2}{2^N}$$

9. Vamos finalizar esta secção apresentando uma outra forma de obter o resultado anterior (se bem que para um caso mais genérico).

   Retomemos a igualdade apresentada acima para caracterizar a soma de uma progressão geométrica:

   $$x+x^2+x^3+\cdots+x^n=\frac{x-x^{n+1}}{(1-x)}$$

   Derivando ambos os membros (em ordem a x) obtemos

   $$1+2\,x+3\,x^2+\cdots n\,x^{n-1}=\frac{(1-(n+1)\,x^n)\,(1-x)+(x-x^{n+1})}{(1-x)^2}$$

   O que, simplificando, vem

   $$\sum_{i=1}^{n} i\,x^{i-1}=\frac{1+n\,x^{n+1}-(n+1)\,x^n}{(1-x)^2}$$

   Multiplicando ambos os membros por x, temos o resultado que pretendemos:

   $$\sum_{i=1}^{n} i\,x^{i}=x\,\frac{1+n\,x^{n+1}-(n+1)\,x^n}{(1-x)^2}$$

   Verifique (fazendo $x=\frac{1}{2}$) o resultado obtido na alínea anterior.