Ficha 2

Algoritmos e Complexidade

Análise de tempo de execução

1 Análise de programas com ciclos

1. Considere os seguintes dois algoritmos alternativos para a procura de um elemento num vector. Note que o segundo algoritmo só pode ser usado sobre vectores *ordenados* (de forma crescente).

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

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

   (a) Para cada um dos algoritmos, calcule o número de operações primitivas de acesso ao vector de entrada executadas, em função do tamanho desse vector.

   (b) Efectue a análise assimptótica do tempo de execução no melhor e no pior caso de ambas as versões, dizendo claramente a situação a que corresponde cada um dos casos.

   (c) Com base na resposta à questão anterior, poder-se-á afirmar que o segundo algoritmo é melhor do que o primeiro? Justifique.

   (d) Considere ainda o seguinte algoritmo alternativo (pesquisa *binária*):

```
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[m]==x) return m;
    else return (-1);
}
```

Repita a análise assimptótica do tempo de execução no melhor e no pior caso para este algoritmo, e compare-o com os anteriores em termos de eficiência.

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

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

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

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

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

(a) Para cada uma das soluções apresentadas, efectue uma contagem do número de vezes que as operações primitivas contidas no corpo do ciclo são executadas.

(b) Efectue agora a análise assimptótica do tempo de execução no pior caso de ambas as versões, considerando que o tamanho do input é *o número de* bits *necessários para representar os números inteiros em causa*. Recorde que, por exemplo, os números cuja representação requer 5 bits são $\{16, \ldots, 31\}$.

3. Considere os seguintes algoritmos de ordenação:

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);
}
```

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

Para cada um,

(a) Determine o melhor e o pior caso para o número de trocas (chamadas à função `swap`).

(b) Determine o melhor e o pior caso para o número de comparações entre elementos do *array*.

(c) Qual das operações poderá ser considerada representativa do ponto de vista do tempo de execução do algoritmo? Efectue a análise assimptótica do tempo de execução no melhor e no pior caso.

(d) O princípio em que se baseia o algoritmo *Bubble sort* (e o invariante do ciclo exterior) é o mesmo dos algoritmos anteriores, mas o i-ésimo elemento do vector é agora colocado na posição final i através de uma sequência de operações `swap` entre posições adjacentes do vector. O algoritmo incorpora uma optimização que consiste em parar quando a colocação de um elemento é conseguida sem que seja necessário um único `swap`, o que significa que o vector está já ordenado.

O algoritmo pode ser mais optimizado. Para o vector $[10, 20, 30, 40, 80, 70, 60, 50]$ obtemos na primeira iteração do ciclo exterior $[10, 20, 30, 40, 50, 80, 70, 60]$, sendo que o último `swap` efectuado nesta iteração envolveu as posições 4 (80) e 5 (50). Isto significa que os elementos nas posições $0 \ldots 4$ estão já ordenados, podendo por isso ser deixados de fora da próxima iteração.

Implemente esta optimização e repita a análise de melhor e pior caso para o algoritmo optimizado.

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

```
int repetidos (int v[], int N) {
    int i, j, rep = 0;
    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 no que respeita ao número de comparações (entre elementos do vector).

(b) Quantas operações são executadas em cada caso? Efectue a análise assimptótica do tempo de execução em ambos os casos.

2 Análise de programas recursivos e relações de recorrência

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. 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);
  }
}
```

   (a) Escreva uma relação de recorrência que exprima a complexidade deste algoritmo (por exemplo, em função do número de linhas impressas).

   (b) Desenhe a árvore de recursão do algoritmo e obtenha a partir dessa árvore um resultado sobre a sua complexidade assimptótica.

3. Considere a seguinte definição em `Haskell` do algoritmo de ordenação por inserção (*insertion sort*) em listas.

```
isort :: (Ord a) => [a] -> [a]
iSort [] = []
isort (h:t) = insert h (isort t)
    where insert y [] = [y]
          insert y (x:xs) | y <= x    = y:x:xs
                          | otherwise = x:(insert y xs)
```

   Esta definição pode ser convertida numa função em C que ordena um vector:

```
void isort (int v[], int N) {
     int i; int t;
     if (N>0) {
          isort (v+1, N-1);
          i = 0;
          t = v[0];
          while (i<N-1 && v[i]<t) {
               v[i] = v[i+1];
               i++;
          }
          if (i>0) v[i] = t;
     }
}
```

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

(b) Para esses casos, apresente uma relação de recorrência que traduza o número de comparações entre elementos do vector em função do tamanho do vector.

4. Considere o seguinte algoritmo para o cálculo de números de Fibonacci:

```
int fib (int n)
{
  if (n==0 || n==1) return 1;
  else return fib(n-1) + fib(n-2);
}
```

Apesar de traduzir exactamente a definição da sequência de números de Fibonacci, este algoritmo é muito ineficiente (de tempo exponencial). Assuma que as operações aritméticas elementares se efectuam em tempo $\mathcal{O}(1)$.

(a) Escreva uma recorrência que descreva o comportamento temporal do algoritmo. Desenhe a respectiva árvore de recursão para $n = 5$.

(b) Efectue uma análise assimptótica do tempo de execução deste algoritmo.

(c) Escreva em **C** um algoritmo alternativo mais eficiente. Analise o seu tempo de execução.

3 Análise do tempo médio

1. Relembre as funções `xlsearch`, `lsearch` e `bsearch` apresentadas na secção 1. Para cada uma dessas funções apresente o número médio de acessos ao array.

   Assuma que os `N` elementos do array são todos diferentes, e que são representados por `W` bits (ou seja, existem `2^W` inteiros diferentes).

2. 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 da função `inc`, 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.

3. Relembre a função `particao` e as funções em que ela foi usada (`qSort` e `kesimo`). Assumindo que na função `kesimo` todos os valores de retorno possíveis são equiprováveis (i.e., se o array tiver `N` elementos a probabilidade de cada valor é `1/N`), calcule o tempo médio das funções `qSort` e `kesimo`).

4 Análise Amortizada

1. Relembre a função `inc` da secção anterior. Use (os três métodos de) análise amortizada para mostrar que o tempo médio de execução desta função é constante.

2. Uma definição alternativa de *Stacks* às clássicas representações em *array* ou em lista ligada, consiste no uso de arrays dinâmicos. Esta solução tem a simplicidade da implementação em *array*, aliada às vantagens de usar uma estrutura dinâmica.

```
typedef struct {
    int size, used;
    int *table;
} DynTable;
```

   As operações de adição e remoção de um elemento (por exemplo nas *stacks*, as operações de *push* e *pop*) deverão testar a capacidade usada da tabela e, em certos casos, realocar os elementos da tabela:

   - ao acrescentar um elemento a uma tabela cheia (`size == used`) deve-se começar por realocar os elementos da tabela para uma com o dobro da capacidade.
   - Ao remover um elemento de uma tabela, se ela passar a estar apenas a 25% da sua capacidade, devem-se realocar os elementos para uma tabela com metade da capacidade.

   Usando análise amortizada, mostre que esta solução tem custo amortizado constante das operações de inserção e remoção.

3. 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 excepto o último, faz-se sucessivamente *pop* e *push* na *stack* B. Faz-se depois *pop* do último, 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 $\mathcal{O}(1)$. Faça isto das seguintes formas.

i. Defina a sequência de N operações que, partindo de uma queue vazia, tem o maior custo. Calcule o custo médio de cada operação nessa sequência (análise agregada).

ii. Apresente estimativas para o custo amortizado de cada operação de forma que para qualquer sequência de operações o somatório dos custos amortizados seja maior do que o custo real dessa sequência de operações (método contabilistico).

iii. Defina uma função de potencial que permita concluir que o custo amortizado de cada operação é $\mathcal{O}(1)$. Baseado nesse potencial defina o custo amortizado de cada uma das operações de inserção e remoção de um elemento na *queue* (método do potencial).

4. Uma `quack` é uma estrutura que combina as funcionalidades de uma `queue` com as de uma `stack`. Pode ser vista como uma lista de elementos em que são possíveis três operações:

   - `push` que adiciona um elemento;
   - `pop` que remove o último elemento inserido;
   - `pull` que remove o elemento inserido há mais tempo.

   Apresente uma implementação de *quack*s usando 3 `stacks` garantindo que o custo amortizado de cada uma das três operações é $O(1)$, assumindo que todas as operações das `stacks` são realizadas em tempo constante.

   Justifique a sua implementação usando uma das 3 formas estudadas de análise amortizada.

A Exercícios Adicionais

1. 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* necessários para representar o expoente (Sugestão: use como inspiração as funções apresentadas na Secção 1 para calcular o produto de dois números).

2. 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?

3. 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, 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).

4. O algoritmo *Insertion sort* estudado nas aulas teóricas pode também ser implementado com base na operação `swap`, como se segue.

   Insertion-sort

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

   (a) Determine o melhor e o pior caso para o número de trocas (chamadas à função `swap`).

   (b) Determine o melhor e o pior caso para o número de comparações entre elementos do *array*.

   (c) Efectue a análise assimptótica do tempo de execução no melhor e no pior caso.

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

```
void msort (inv v[], int N) {
int *aux = (int *) malloc (N*sizeof(int));
```

```
    msortAux (v, aux, 0, N-1);
    free (aux);
}

void msortAux (int v[], int aux [], int a, int b) {
int m;

    if (a<b) {
        m = (a+b)/2;
        msortAux (v, aux, a, m);
        msortAux (v, aux, m+1, b);
        merge (x, aux, a, m, b);
       }
}
```

(a) Defina a função `merge` usada acima e que funde duas porções de um vector (uma com índices $[a \ldots m]$ e outra com índices $[m+1 \ldots b]$) usando um vector `aux` como auxiliar.

Garanta que o tempo de execução dessa função é $\Theta(N)$ em que $N = b - a + 1$, i.e., a soma dos tamanhos dos dois vectores a serem fundidos.

(b) Apresente uma relação de recorrência que traduza o tempo de execução de `msort` (ou equivalentemente de `msortAux`), em função do tamanho do vector argumento.

Apresente ainda uma solução dessa recorrência.

6. O algoritmo de ordenação *Quicksort* pode ser implementado em C da seguinte forma:

```
void qSort (int v [], int N) {
   qSortAux (v, 0, N-1);
}

void qSortAux (int v[], int a, int b) {
int p;

    if (a<b) {
        p = particao (v,a,b);
        qSortAux (v,a,p-1);
        qSortAux (v,p+1,b);
    }
}
```

A função `particao` reorganiza o vector `v[a..b]` e retorna um índice `p` de tal forma que, após a sua terminação,

$$\forall_{a \leq k < p}(v[k] < v[p]) \wedge \forall_{p < k \leq b}(v[k] \geq v[p])$$

(a) Complete a seguinte definição da função `particao`.

```
int particao (int v[], int a, int b)}
int i, j;
    i = ...; j = ...
    while (...) {
        ...
        j++;
       }
    swap (v,i,b);
    return i;
}
```

Para isso use o seguinte predicado como invariante do ciclo `while`:

$$\forall_{a \leq k < i}(v[k] < v[b]) \wedge \forall_{i \leq k < j}(v[k] \geq v[b])$$

(b) Mostre que o tempo de execução da função `particao` é linear no tamanho do vector ($\Theta(N)$).

(c) Assumindo que os valores do vector são tais que o valor da função `particao (v,a,b)` é sempre de `(a+b)/2`, apresente uma relação de recorrência que traduza o tempo de execução do *quick-sort* em função do tamanho do *array*.

Apresente ainda uma solução dessa recorrência.

(d) Em que casos é que o valor da função `particao (v,a,b)` é sempre igual a `b`? Qual o tempo de execução desta função para esse caso?

7. Considere agora o problema de, num *array* não ordenado (e sem repetições) determinar o k-ésimo menor elemento. Uma das soluções mais eficientes consiste em aproveitar a função de partição do algoritmo de *quick-sort* apresentada acima.

```
int kesimo (int v[], int N, int k){
int a=0,b=N-1,p=-1;

    while (p!=k) {
        p = particao (b,a,b);
        if p<k b = p-1;
        else   a = p+1;
       }
return v[p];
}
```

Assumindo que os valores do vector são tais que o valor da função `particao (v,a,b)` é sempre de `(a+b)/2`, apresente uma relação de recorrência que traduza o tempo de execução do *kesimo* em função do tamanho do *array*.

Apresente ainda uma solução dessa recorrência.