

T1. Tempo de Execução de Algoritmos Iterativos

Contabilização de operações primitivas

Iremos:

1. identificar as operações primitivas que executam em *tempo constante*, i.e. tempo que não depende do tamanho do input
2. atribuir *custos (tempos) abstractos* c_1, c_2, \dots a estas operações primitivas, de forma independente de qualquer mecanismo de execução concreto
3. calcular o custo abstracto total do algoritmo em função do *tamanho do input* – $T(N)$

Exemplo: contagem de ocorrências num array

```
int conta (int k, int v[], int N) {  
    int i = 0, r = 0;           c1      1  
    while (i < N) {             c2      N+1  
        if (v[i] == k) r++;      c3      N  
        i++;                    c4      N  
    }  
    return r;                   c5      1  
}
```

Note-se que

- c_1 corresponde ao custo das duas operações de inicialização;
- a condição do ciclo é uma operação primitiva, avaliada $N+1$ vezes (falso na última vez), com custo c_2

Temos: $T(N) = c_1 + c_2(N + 1) + c_3N + c_4N + c_5$ ou simplificando:

$$T(N) = (c_2 + c_3 + c_4)N + c_1 + c_2 + c_5$$

Polinómio de primeiro grau em N : crescimento linear

Exemplo: identificação de duplicados num array

```
void dup (int *v, int a, int b) {  
    int i, j;  
    for (i=a ; i<=b ; i++)           c1           N+1  
        for (j=a ; j<=b ; j++)       c2           N*(N+1)  
            if (i!=j && v[i]==v[j])   c3           N*N  
                printf("%d igual a %d\n", i, j);  
}
```

- A função procura elementos repetidos entre os índices a e b. Sendo assim, o tamanho do input será dado por $N = b - a + 1$.
- c1 é o custo agregado da condição $i < b$ (testada $N+1$ vezes) e do incremento $i++$ (feito N vezes)
- O ciclo exterior executa N iterações, e em cada iteração, o ciclo interior executa N iterações. Sendo assim:

— O condicional (custo c3) é executado no total N^2 vezes

Logo:

$T(N) = c_1(N + 1) + c_2N(N + 1) + c_3N^2$ ou simplificando:

$$T(N) = (c_2 + c_3)N^2 + (c_1 + c_2)N + c_1$$

Polinómio de segundo grau em N : crescimento quadrático

Exemplo: identificação de duplicados num array

A versão anterior imprime cada par duas vezes! Pode ser optimizada:

```
void dup2 (int v[], int a, int b) {
```

int i, j;		
for (i=a ; i<b ; i++)	c1	N
for (j=i+1 ; j<=b ; j++)	c2	Soma S1
if (v[i]==v[j])	c3	Soma S2
printf("%d igual a %d\n", i, j);		
}		

Sendo mais eficiente, esta versão é mais difícil de analisar, uma vez que, *para cada valor de i fixado pelo ciclo exterior, o ciclo interior executará um número diferente de vezes!*

A tabela seguinte detalha os valores tomados por j e o número de iterações do ciclo interior, para cada valor de i:

i=a	j = a+1 ... b	b-a iterações
i=a+1	j= a+2 ... b	b-a-1 iterações
...
i (arbitrário)	j = i+1 ... b	b - (i+1) +1 = b-i
...
i=b-1	j = b	1 iteração

Somando todas estas iterações, obtemos $S_2 = \sum_{i=a}^{b-1} b - i$ e também $S_1 = \sum_{i=a}^{b-1} b - i + 1$.

Admitamos sem perda de generalidade que $a = 1$ e $b = N$. Então

$$S_2 = \sum_{i=1}^{N-1} (N - i) = (N - 1)N - \frac{(N-1)N}{2} = \frac{1}{2}N^2 - \frac{1}{2}N$$

$$S_1 = \sum_{i=1}^{N-1} (N - i + 1) = S_2 + N - 1$$

logo:

$$T(N) = c_1N + c_2S_1 + c_3S_2$$

$$T(N) = k_2N^2 + k_1N + k_0$$

Polinómio de segundo grau em N: crescimento quadrático

(veremos que não é muito importante calcular os coeficientes k_0 a k_2)

Análise Assimptótica

Numa função polinomial como $T(N) = k_2N^2 + k_1N + k_0$,

para *inputs de tamanho elevado* o efeito dos termos de menor grau é anulado face ao crescimento do termo de maior grau.

E de facto, o interesse da constante multiplicativa k_2 é também pequeno: na *análise assintótica* interessamo-nos apenas pela *ordem de crescimento* do tempo de execução dos algoritmos.

Escreveremos:

$$T(N) = \Theta(N^2)$$

Se o algoritmo A1 é *assintoticamente melhor* do que A2, será melhor escolha do que A2 *excepto para inputs muito pequenos*.

Notação O (“big oh”)

Para uma função g não-negativa de domínio \mathbf{N} , define-se $O(g)$ como o seguinte *conjunto de funções*:

$$O(g) = \{f \mid \text{existem } c, n_0 > 0 \text{ tais que } \forall n \geq n_0. 0 \leq f(n) \leq cg(n)\}$$

Para $n \geq n_0$, g é um limite superior de f a menos de um factor constante

Exemplos:

- $3n^2 + 7n \in O(n^2)$

- $4n - 5 \in O(n^2)$
- $7n^3 - 2n \notin O(n^2)$

Notação Ω (Omega)

Para uma função g não-negativa de domínio \mathbb{N} , define-se $\Omega(g)$ como o seguinte *conjunto de funções*:

$$\Omega(g) = \{f \mid \text{existem } c, n_0 > 0 \text{ tais que } \forall n \geq n_0. 0 \leq cg(n) \leq f(n)\}$$

| Para $n \geq n_0$, g é um limite inferior de f a menos de um factor constante

Exemplos:

- $3n^2 + 7n \in \Omega(n^2)$
- $4n - 5 \notin \Omega(n^2)$
- $7n^3 - 2n \in \Omega(n^2)$

Notação Θ (Theta)

Para uma função g não-negativa de domínio \mathbb{N} , $\Theta(g) = O(g) \cap \Omega(g)$

| Se $f \in \Theta(g)$, então para $n \geq n_0$ f tem um comportamento “igual” ao de g , a menos de factores constantes

Exemplos:

- $3n^2 + 7n \in \Theta(n^2)$
- $4n - 5 \notin \Theta(n^2)$
- $7n^3 - 2n \notin \Theta(n^2)$

Para os exemplos acima podemos escrever:

- Contagem de ocorrências num *array*: $T(N) = \Theta(N)$
- Identificação de duplicados num *array*: $T(N) = \Theta(N^2)$

Funções Úteis em Análise de Algoritmos

As funções tipicamente utilizadas são as *logarítmicas*, *polinomiais*, e *exponenciais*. Alguns factos envolvendo estas funções:

A base das funções logarítmicas é irrelevante assintoticamente (desde que > 1) :

$$\left| \log_b n = O(\log_a n) \text{ se } a, b > 1 \right.$$

Já no caso das funções polinomiais, qualquer polinómio é limitado superiormente por qualquer outro polinómio de maior grau:

$$\left| n^b = O(n^a) \text{ se } b \leq a \right.$$

O mesmo acontece com as funções exponenciais relativamente à base:

$$\left| b^n = O(a^n) \text{ se } b \leq a \right.$$

Por outro lado, qualquer função logarítmica é limitada superiormente por qualquer função polinomial:

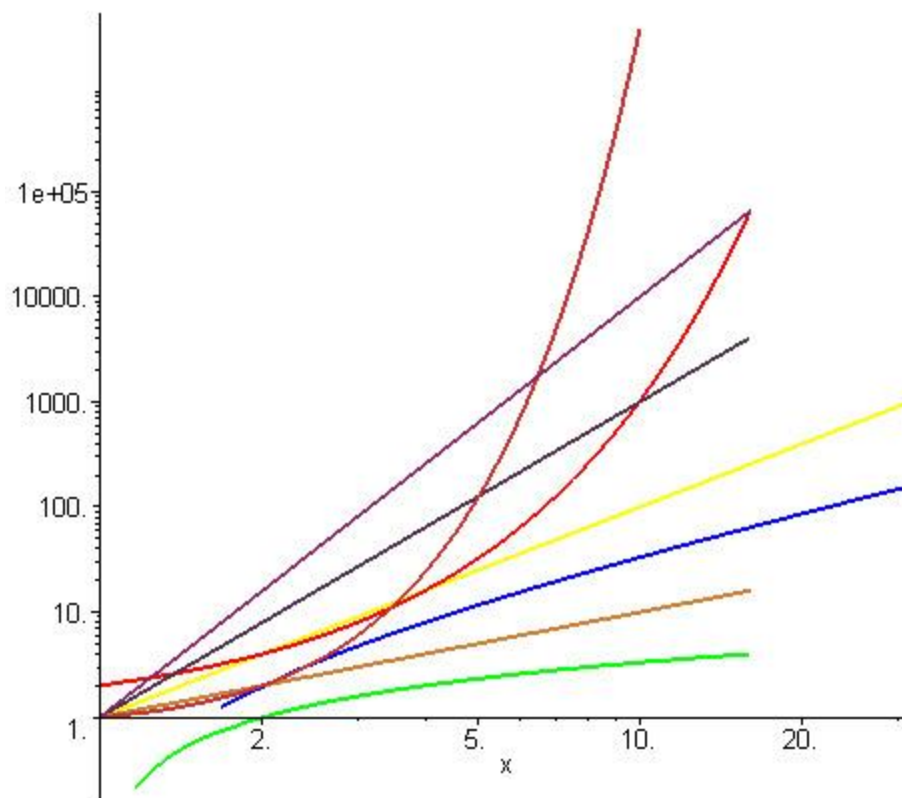
$$\left| \log_b n = O(n^a) \right.$$

e qualquer função polinomial é limitada superiormente por qualquer função exponencial de base > 1 :

$$\left| n^b = O(a^n) \text{ se } a > 1 \right.$$

Crescimento de Algumas Funções Típicas

Tempo (μs)		$33n$	$46n \lg n$	$13n^2$	$3.4n^3$	
Tempo assimp.		n	$n \lg n$	n^2	n^3	2^n
Tempo de execução por tamanho do input	$n=10$.00033s	.0015s	.0013s	.0034s	.001s
	$n=100$.003s	.03s	.13s	3.4s	$4 \cdot 10^{14}s$
	$n=1000$.033s	.45s	13s	.94h	séculos
	$n=10000$.33s	6.1s	22m	39 dias	...
	$n=100000$	3.3s	1.3m	1.5 dias	108 anos	...
Tamanho máx. do input para	1s	$3 \cdot 10^4$	2000	280	67	20
	1m	$18 \cdot 10^5$	82000	2200	260	26



$\log x$, funções polinomiais de diversos graus (rectas, incluindo $n \lg n$), 2^x , e x !

Identificação de Operações Relevantes

- Na prática, para analisar assintoticamente um algoritmo não é necessário considerar custos abstractos associados às operações primitivas: basta contar o número de vezes que as operações são executadas.
- De facto não é sequer necessário contar *todas* as operações primitivas. Basta identificar as que são relevantes para a análise assintótica.

Dadas duas operações op1 e op2 de um programa, se o número de execuções de op1 não for assintoticamente superior ao número de execuções de op2, então op1 pode ser descartada para efeitos de análise (assimptótica) de tempo de execução

Exemplo:

```
void dup2 (int v[], int a, int b) {  
    int i, j;  
    for (i=a ; i<b ; i++)  
        for (j=i+1 ; j<=b ; j++)           Soma S1  
            if (v[i]==v[j])                 Soma S2  
                printf("%d igual a %d\n", i, j);  
}
```

Neste algoritmo basta considerar o condicional (corpo do ciclo interior) como única instrução assintoticamente relevante. O tempo de execução pode ser analisado calculando simplesmente: $S_2 = \sum_{i=a}^{b-1} b - i = \Theta(N^2)$.

Apesar de a condição $j \leq b$ ser avaliada mais vezes ($S_1 = S_2 + N - 1$) do que o condicional (S2), assintoticamente é equivalente considerar qualquer uma das duas para efeitos de análise.

*Na prática contamos o **número de operações de comparação** efectuadas (entre elementos do array), o que é bastante intuitivo: as comparações são as operações fundamentais deste algoritmo.*

EXERCÍCIO: Calcule o tempo de execução da seguinte versão alternativa desta função:

```
void dup3 (int v[], int a, int b) {  
    int i, j;  
    for (i=a+1 ; i<=b ; i++)  
        for (j=a ; j<i ; j++)  
            if (v[i]==v[j])  
                printf("%d igual a %d\n", i, j);  
}
```

$$T_{==}(n) = \sum_{i=a+1}^b i - 1 - a + 1 = \sum_{i=a+1}^b i - a = \sum_{i=a+1}^b \sum_{j=a}^{i-1} 1$$

$$T_{j<i}(n) = \sum_{i=a+1}^b \sum_{j=a}^i 1$$

Sem perda de generalidade, consideramos $a = 0$ e $b = n - 1$

$$T_{==}(n) = \sum_{i=1}^{n-1} i = \frac{(n-1)*n}{2} = \theta(n^2)$$

$$T_{j<i}(n) = \sum_{i=1}^{n-1} i + 1 = \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 = \frac{(n-1)*n}{2} + n - 1 = \theta(n^2)$$

$$T(n) = \theta(n^2)$$

Tamanho e Representação

A noção apropriada de tamanho de um número corresponde ao número de caracteres necessários para o escrever: o tamanho de 3500 é 4 em notação decimal.

Um inteiro n em notação decimal ocupa aproximadamente $\log_{10} n$ dígitos; em notação binária (representação em máquina) ocupa $\log_2 n$ dígitos.

EXEMPLO:

Problema: Dado um inteiro positivo n , haverá dois inteiros $j, k > 1$ tais que $x = jk$? (i.e, será x um número primo ou não?)

Considere-se o seguinte algoritmo de “força bruta”:

```
found = 0;
j = 2;
while ((!found) && j < x) {
    if (x mod j == 0) found = 1;
    else j++;
}
```

Este algoritmo executa em tempo $O(x)$, no entanto trata-se de um problema famoso pela sua dificuldade (e é por isso relevante em muitos algoritmos criptográficos).

Observe-se:

- se o algoritmo executa em tempo linear $O(x)$,
- e a representação de x utiliza pelo menos $N = \log_k x$ dígitos (ou bits),
- então $x = O(k^N)$,
- e o algoritmo executa por isso em tempo $T(N) = O(k^N)$.

Ou seja: *um algoritmo de tempo aparentemente linear é de facto de tempo exponencial no tamanho da representação do número!*

É importante identificar correctamente a noção adequada de tamanho do input de um problema, uma vez que disso depende a sua classificação como fácil ou difícil!

Exercício

Um dos algoritmos de *ordenação* mais simples é conhecido por *selection sort*. A ideia é, em cada iteração j do ciclo exterior (ciclo for), colocar na posição de índice j do array o

j -ésimo menor elemento. O ciclo interior (while) determina o índice do menor elemento ainda não colocado na sua posição final, e a função swap é usada para trocar este elemento com o que se encontra na posição j .

```
void swap(int t[],int i,int j) {
    int tmp = t[i];
    t[i] = t[j];
    t[j] = tmp;
}

void selectionSort(int A[], int n) {
    int i, j, min, k;

    for (j=0; j < n-1; j++) {
        min = j;
        i = j+1;
        while (i < n) {
            if (A[min] > A[i]) min = i;
            i++;
        }
    }
}
```

```
        if (j != min) swap (A, j, min);  
    }  
}
```

Note que a função swap executa em *tempo constante*, i.e. o seu tempo de execução não depende do tamanho do array. Escreveremos $T_{\text{swap}}(N) = \Theta(1)$.

Tendo isto em conta, identifique uma ou mais operações relevantes e analise assintoticamente o tempo de execução da função.