ISEL | DEETC

Algoritmos e Estruturas de Dados

Semestre de Inverno de 2009-2010

1.ª Série de Exercícios

Autores:
30896 – Ricardo Canto
31401 – Nuno Cancelo
33595 – Nuno Sousa

Indície

Enunciado

Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores
Algoritmos e Estruturas de Dados
Semestre de Inverno 2009/10
Primeira série de exercícios

Observações:

- Data de entrega: **19 de Outubro de 2009**.
- No contexto desta série, o triplo (v, l, r) representa o *subarray* do *array* v, compreendido entre os índices l e r inclusivé.

1 Introdução

1. Realize o método estático

```
public static int removeOdd(int[] v, int count);
```

Este método retira os inteiros ímpares da sequência representada pelos primeiros `count` inteiros do *array* `v`. A sequência resultante fica contida de forma contígua nas primeiras posições do *array* `v`. O valor retornado pelo método é a dimensão da sequência resultante.

2. Realize o método estático

```
public int countEqualTo(int[] v, int l, int r, int a)
```

que retorna o número de ocorrências do inteiro `a` no *subarray*(v, l, r), que está ordenado de forma crescente. O custo assimptótico deve ser $O(\log N)$, onde N é a dimensão do *subarray*.

Tenha em consideração que o número de ocorrências pertence a $O(N)$.

3. Realize o método estático

```
public static boolean isMaximumSubArrayGivenIndex(int[] v, int l, int r, int i)
```

que retorna *true* se e só se $l \leq i \leq r$ e a soma dos elementos do *subarray* (v, l, r) tiver o maior valor possível para a soma de qualquer *subarray* de v contendo o índice i.

4. Realize o método estático

```
public static IntTriple getMaximumSubArrayGivenIndex(int[] v, int i)
```

que retorna o triplo de inteiros (l, r, s) tal que o resultado da avaliação de `isMaximumSubArrayGivenIndex(v,l,r,i)` seja `true` e $s = \sum_l^r v[i]$. O custo assimptótico deve ser $O(N)$, onde N é a dimensão do *array*.

Exemplo

Considere o *array* $v = \{12, -2, -24, 26, -3, -17, -18, 20, 16, -10, 12, -2, 10, -15\}$. O triplo resultante da chamada do método com $i = 3$ é $(3, 12, 34)$. A chamada com $i = 7$ resulta no triplo $(7, 12, 46)$.

5. Considere o método

```
public static IntTriple getMaximumSubArray(int[] v)
```

que retorna o triplo (l, r, s), tal que (v, l, r) seja um *subarray* de v com maior valor para a soma dos seus elementos e s seja o valor dessa soma. Sendo N a dimensão do *array*,

5.1. Realize uma implementação usando um algoritmo com custo assimptótico $\Theta(N^2)$. Sugestão: pesquisa exaustiva utilizando o método `getMaximumSubArrayGivenIndex`.

5.2. Realize uma implementação usando um algoritmo com custo assimptótico $\Theta(N \log N)$. Sugestão: divisão do problema em dois subproblemas e utilização do método `getMaximumSubArrayGivenIndex`.

5.3. Realize uma implementação usando um algoritmo com custo assimptótico $\Theta(N)$. Sugestão: percurso linear usando as seguintes observações:

- O primeiro e último elemento do *subarray* solução são não negativos.
- Se o *subarray* (v, l, r) tem soma negativa, então o *subarray* (v, l, r'), para qualquer $r' > r'$, não é solução.

Avalie experimentalmente as implementações anteriores.

2 Análise de desempenho

1. Considere o seguinte algoritmo

```
METHOD(v, l, r, s)
if r < l
    then return s
m ← (l + r)/2
s ← s + v[m]
s ← METHOD(v, l, m − 1, s)
s ← METHOD(v, m + 1, r, s)
returns;
```

Indique o número máximo de chamadas ao método METHOD existentes na execução do algoritmo sobre um *sub-array* de dimensão $2^N - 1 = r - l + 1$.

2. Prove que $O(N) + O(N \log N) = O(N \log N)$.

3. Sejam $f, g : \mathbf{N}_0 \longrightarrow \mathbf{R}^+$ funções. Indique se cada uma das seguintes proposições é falsa ou verdadeira.

   3.1. $f = O(g)$ então $g = O(f)$;

   3.2. $f + g = \Theta(min(f, g))$;

   3.3. $f = O(f^2)$;

   Justifique as respostas, apresentando a prova da proposição ou um contra-exemplo.

4. Resolva as seguintes recorrências, usando notação assimptótica para representar o resultado na forma de uma função explícita.

   4.1. $C(n) = C(n/3) + 1$

   4.2. $C(n) = 2C(n/2) + n \log n$.

ISEL, 28 de Outubro de 2009.

Resolução

Exercício 1:

Na análise do enunciado não nos foi totalmente claro o que era pretendido, assim sendo optámos por desenvolver duas hipóteses de resolução do enunciado.

A ambiguidade do enunciado refere-se ao seguinte paragrafo:

"A sequência resultante fica contida de forma contigua nas primeiras posições do array v. O valor retornado pelo método é a dimensão da sequência resultante.", no que diz respeito ao restante do array.

Abordagem 1:

Podemos entender que o array final, após a remoção dos valores ímpares, é contigua nas primeiras (count – remoções) posições, não interessando como está o restante array.

```
public static int removeOdd(int[] v, int count){
        int dim=0;
        for (int idx=0;idx<count;idx++)
         if (v[idx]%2 == 0) v[dim++]=v[idx];
        return dim;
}
```

Após a execução deste método, obtemos um array com as primeiras *dim* posições com os valores pares e a partir dai até ao final do array com o que estava nas respectivas posições.

Abordagem 2:

Nesta abordagem, após a remoção dos ímpares, movemos as posições do array o numero de posições removidas, ficando todo ele contiguo.

```
public static int removeOdd(int[] v, int count){
        int dim=0;
        for (int idx=0;idx<v.length;idx++){
                if (idx<count){
                        if (v[idx]%2 == 0)v[dim]=v[idx];
                }else{
                        v[dim++]=v[idx];
                }
        }
        return dim;
}
```

Nesta implementação, o array está todo contiguo até a posição dim estando os restantes posições com lixo.

Exercício 2:

Neste exercício, compreendemos depressa o que o método deveria fazer, no entanto a forma como o haveríamos de foi mais complexo uma vez que pretendíamos que o algoritmo fosse mais eficiente do que propriamente eficaz.

Uma informação bastante útil foi o facto de se indicar que o array estava ordenado, que nos permite executar uma pesquisa binária para obter os indicies.

Uma abordagem utilizada foi efectuar uma pesquisa binária que encontra-se o primeiro elemento a no array e a partir dai contabilizar o numero de elementos do array. Porém depressa compreendemos que este método era eficaz e não eficiente, uma vez que por um lado estávamos a contabilizar ocorrências iguais, que num conjunto pequeno até poderia ser mais rápido, mas um conjunto suficientemente grande estaríamos de facto a desperdiçar tempo.

Por esta altura optámos por obter os indicies de ocorrência para a direita e para a esquerda, ou seja , encontrar o primeiro elemento igual ao valor passado e encontrar o último elemento, e daí fazer a diferença para obter o número de ocorrências. Para este efeito concretizamos dois métodos, o *getFirstOccurrence* e o *getLastOccurrence*, cuja a única diferença se encontra no facto de onde se verifica a igualdade do conteúdo com o valor passado.

```
public static int getFirstOccurrence(int[] v, int l, int r, int a) {
        if (l>r) return -1;
        int mid=(l + r) / 2,val=0;
        if (v[mid] < a)
                val=getFirstOccurrence(v, mid + 1, r, a);
        else
                val=getFirstOccurrence(v,l, mid - 1, a);
        if(val==-1 && v[mid] == a ) return mid;
return val;
}
public static int getLastOccurrence(int[] v, int l, int r, int a) {
        if (l>r) return -1;
        int mid=(l + r) / 2, val=0;
        if (v[mid] > a)
                val=getLastOccurrence(v, l,mid - 1, a);
        else
                val=getLastOccurrence(v, mid + 1,r, a);
        if(val==-1 && v[mid] == a ) return mid;
return val;
}
```

Assim procedemos à realização do método solicitado, esta solução permite-nos realizar um método que tem o problema dividido em dois e que os executa um de cada vez. Permite também promover a execução dos dois métodos somente quando haja o elemento pretendido no array.

```
public static int countEqualTo(int[] v, int l, int r, int a) {

        int left=0,right=0;

        if ((left=getFirstOccurrence(v, l, r, a))!=-1){

                right=getLastOccurrence(v, l, r, a);

                return (right- left +1);

        }

        return -1;

}
```

Exercício 3:

Neste exercício foi já necessário alguma ginástica mental para proceder uma implementação mais correcta.

Verificando que as alíneas seriam semelhantes e ao mesmo tempo diferentes, optámos por ter um esforço adicional na implementação deste método de forma que o pudéssemos executar no futuro sem ter repetir os mesmos procedimentos.

Desde o exercício anterior tornou-se evidente que a divisão do problema em subproblemas tornaria a análise, o planeamento e implementação mais eficiente.

Assim, para este exercício observámos que teríamos que verificar para cada um dos lados se o valor era o maior possível, e dado o retorno booleano bastava-nos fazer a comparação por entre os valores. Então pensámos em duas implementações, uma que executava o solicitado e outra, que apesar se tornar mais lenta no contexto deste exercício, que permite reutilização do código para executar outro exercício.

Abordagem 1:

São implementados dois métodos, *isMaximunSubArrayGivenIndexRight* e *isMaximunSubArrayGivenIndexLeft,* que retornam a indicação se são os valores "correctos" para o que é pretendido.

```
public static boolean isMaximumSubArrayGivenIndexLeft(int[] v, int l, int i) {
        int l_tmp = i;

        for (int idx = i - 1, sum = 0; idx >= l; --idx) {
                if (v[idx] > 0 && ((sum + v[idx]) > 0)) {
                        l_tmp = idx;
                        sum = 0;
                } else {
                        sum += v[idx];
                }
        }
        return (l == l_tmp);

}
```

```
public static boolean isMaximumSubArrayGivenIndexRight(int[] v, int r, int i) {

        int r_tmp = i;
        for (int idx = i + 1, sum = 0; idx <= r; ++idx) {
                if (v[idx] > 0 && ((sum + v[idx]) > 0)) {
                        r_tmp = idx;
                        sum = 0;
                } else {
                        sum += v[idx];
                }
        }
        return r == r_tmp;
}

public static boolean isMaximumSubArrayGivenIndex(int[] v, int l, int r,int i) {

        return (isMaximumSubArrayGivenIndexLeft(v, l, i) &&
isMaximumSubArrayGivenIndexRight(v, r, i));
}
```

Com esta implementação, assim que o valor à esquerda não for verdadeira, deixa de ser necessário a verificação à direita.

Abordagem 2:

São implementados dois métodos, *MaximumSubArrayIndexLeft* e MaximumSubArrayIndexRight, que é semelhante da implementação 1, diferindo em dois aspectos: Efectua a soma dos valores e Retorna um objecto do tipo IntTriple.

```
public static IntTriple MaximumSubArrayIndexLeft(int[] v, int l, int i) {
        int l_tmp = i,total =0;

        for (int idx = i - 1,sum=0; idx >= l; --idx) {
                if (v[idx] > 0 && ((sum + v[idx]) > 0)) {
                        l_tmp = idx;
                        total += sum + v[idx];
                        sum = 0;
                } else {
                        sum += v[idx];
                }
        }
        return (new IntTriple(l_tmp,0,total));
}

public static IntTriple MaximumSubArrayIndexRight(int[] v, int r, int i) {
        int r_tmp = i,total=0;

        for (int idx = i + 1,sum=0; idx <= r; ++idx) {
                if (v[idx] > 0 && ((sum + v[idx]) > 0)) {
                        r_tmp = idx;
                        total += sum + v[idx];
                        sum = 0;
                } else {
                        sum += v[idx];
                }

        }
        return (new IntTriple(0,r_tmp,total));
}

public static boolean isMaximumSubArrayGivenIndex(int[] v, int l, int r,int i) {
        /*
         *Objectos criados somente para tornar mais legível o código
         */

        IntTriple ll=MaximunSubArrayIndexLeft(v, l, i);
        IntTriple rr=MaximunSubArrayIndexRight(v, r, i);

        return (ll.getLeft()==l && rr.getRight()==r);
}
```

Nesta implementação é introduzida a classe IntTriple, que caracteriza uma os atributos do triplo.

```
	class IntTriple {
		private int left;
		private int right;
		private int sum;

		public IntTriple(int l, int r, int s) {
			left = l;
			right = r;
			sum = s;
		}

		public int getLeft() {
			return left;
		}

		public int getRight() {
			return right;
		}

		public int getSum() {
			return sum;
		}
	}
```

Exercício 4:

Utilizando a segunda abordagem do exercício anterior torna-se simples a implementação do que é solicitado.

```
	public static IntTriple MaximumSubArrayGivenIndex(int[] v, int l, int r, int i) {
		IntTriple left=MaximunSubArrayIndexLeft(v, l, i);
		IntTriple right=MaximunSubArrayIndexRight(v, r, i);
		return (new IntTriple(left.getLeft(), right.getRight(), left.getSum()
+right.getSum() + v[i]) );
	}


	public static IntTriple getMaximumSubArrayGivenIndex(int[] v, int i) {
		return (MaximumSubArrayGivenIndex(v, 0, v.length - 1, i));
	}
```

Exercício 5:

Exercício 5.1:

Para obtermos um custo assimptótico de $\theta(N^2)$ usámos a sugestão dada e efectuámos uma pesquisa exaustiva ao array utilizando o método *GetMaximumSubArrayGivenIndex*:

```
    public static IntTriple getMaximumSubArrayN2(int [] v){
        IntTriple result = new IntTriple();
        for (int i=0; i<v.length; ++i){
            IntTriple aux = Serie01E04.getMaximumSubArrayGivenIndex(v,i);
            if (aux.getSum() > result.getSum())
                result=aux;
        }
        return result;
    }
```

Exercício 5.2:

Neste caso o objectivo é obter um custo assimptótico de $\theta(N*\log(N))$. Obter este custo foi bastante mais difícil que o anterior. Optámos por usar o método auxiliar criado em 4

(*MaximumSubArrayIndexRight)* que foi chamado com um array cada vez menor a partir do indíce mais à esquerda:

```
public static IntTriple MaximumSubArrayNlogN(int [] v, int l, int r){
    IntTriple result = new IntTriple();
    IntTriple aux = new IntTriple();
    while(l<r){
        aux = Serie01E04.MaximunSubArrayIndexRight(v, r, l);
        if (aux.getSum() > result.getSum())
            result = aux;
        l++;
    }
    return result;
}


public static IntTriple getMaximumSubArrayNlogN(int [] v){
    return MaximumSubArrayNlogN(v,0,v.length-1);
}
```

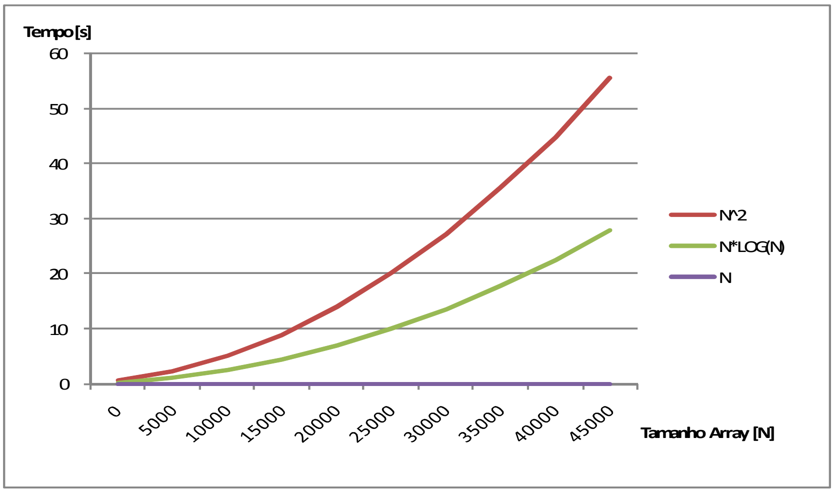
Exercício 5.3:

Para obter um custo de $\theta(N)$ foi necessário estudar muito bem o algoritmo. Apenas corremos o array uma vez (como teria de ser para obter o custo pretendido) e analisamos os valores do mesmo para verificar onde começa e acaba o nosso sub-array.

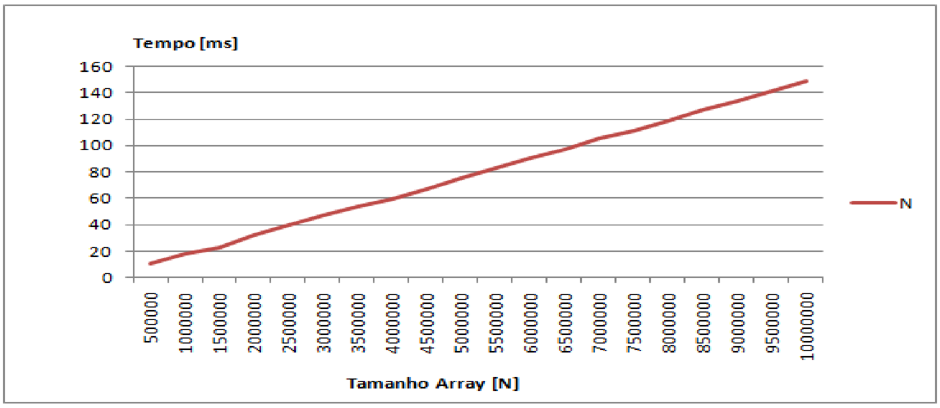
```
public static IntTriple getMaximumSubArrayN(int [] v){
    int sum=0,sum_aux=0, idxl=0, idxr=0, idx_aux=0;
    for (int i=0; i<v.length; ++i){
        sum_aux+=v[i];
        if (sum_aux>sum){
            sum=sum_aux;
            idxl=idx_aux;
            idxr=i;
        } else {
            if (sum_aux<0){
                idx_aux=i+1;
                sum_aux=0;
            }
        }
    }
    return new IntTriple(idxl,idxr,sum);
}
```

Análise experimental:

Para testar as diferenças entre os 3 métodos anteriores efectuámos testes de performance com arrays a começar com $N=5000$ até $N=50000$ de 5000 em 5000 e obtivemos os resultados seguintes:



Os valores obtidos são os esperados sendo que para N<50000 o tempo de execução do 3º método continua a ser demasiado baixo para ser medido em segundos. Para este caso efectuámos o teste a iniciar em $N=500.000$ até $N=10.000.000$ e obtivémos o seguinte resultado:



Podemos verificar facilmente que com custo $\theta(N)$ o método é muito mais eficiente. Com $\theta(N*\log(N))$ é mais eficaz do que $\theta(N^2)$ mas muito menos eficientes que com custo N.

Parte Teórica

Exercício 1

$$\textbf{\textit{METHOD}}(\boldsymbol{v}, \boldsymbol{l}, \boldsymbol{r}, \boldsymbol{s})$$

$\textbf{\textit{if}}\ l<r$

$\quad \textbf{\textit{then return s}}$

$m \leftarrow \frac{(l+r)}{2}$

$s \leftarrow s+v[m]$

$s \leftarrow METHOD(v, l, m-1, s)$

$s \leftarrow METHOD(v, m+1, r, s)$

$\textbf{\textit{return s}}$

Exercício 2:

Prove:

$$O(N)+O(N*LogN)=O(N*LogN)$$

pela definição de O:

$$\{\exists_{c \in R^+}, \exists_{n_0 \in N_0}, \forall_{n>n_0} : f(n)=c*g(n)\}$$

então $O(N)$ e $O(N*LogN)$ verificam as condições da definição.

Efectuando os cálculos:

$$O(N)+O(N*LogN)=O(N*LogN)$$
$$O(N+N*LogN)=O(N*LogN)$$
$$O(max(N, N*LogN))=O(N*LogN)$$

quando é que $N \geq N*LogN$?

$$N \geq N*LogN \Leftrightarrow 1 \geq LogN \Leftrightarrow N \leq 2$$

então para $N>2$, $N*LogN>N$

$$O(max(N, N*LogN))=O(N*LogN)$$
$$O(N*LogN)=O(N*LogN)$$

seja qual for o $c \in R^+$

Exercício 3:

Exercício 3.1:

$f = O(g)\ então\ g = O(f);$

Esta preposição é verdade somente se e só se f e g forem a mesma função, caso contrário é falso.

Ex:

$f(n) = n;\ g(n) = n^2;$

ambas verificam:

$\{\exists_{c \in R^+}, \exists_{n_0 \in N_0}, \forall_{n > n_0} : f(n) = c * g(n)\}$

no entanto:

$f(n) = O(g(n)) \Leftrightarrow n = O(n^2)$

$g(n) = O(f(n)) \Leftrightarrow n^2 \neq O(n)$

Exercício 3.2:

$f + g = \theta(min(f, g));$

Definição:

$\{\exists_{c_1, c_2}, \exists_{n_0 > N_0}, \forall_{n > n_0} : 0 \le c_1 * g(n) \le f(n) \le c_2 * g(n)\}$

Esta preposição é falsa.

Ex.

Se admitirmos que

$f(n) = n;\ g(n) = 1;\ f(n) + g(n) = \theta(min(f(n), g(n))) \Leftrightarrow n + 1 \neq \theta(min(n, 1))$

pois

$\theta(min(n, 1)) = \theta(1)$

Exercício 3.3:

$f = O(f^2)$

Esta preposição será falsa se e só se $0 < f < 1$, uma vez que se:

$f(n) = \frac{1}{n} : f(n) \neq O(f(n)^2)$

Exercício 4:

Exercício 4.1:

$$C(n)=C(\frac{n}{3})+1$$

fazendo uma mudança de variável:

$$m=\log_2(N)\Leftrightarrow 2^m=N$$

$$C(2^m)=C(2^{(m-3)})+O(1)$$

$$C(2^m)=C(2^{(m-4)})+O(1)+O(1)$$

$$C(2^m)=C(2^{(m-5)})+O(1)+O(1)+O(1)$$

…

$$C(2^m)=C(2^{(m-m)})+(m-2)*O(1)$$

$$C(2^m)=C(1)+(m-2)*O(1)$$

$$C(2^m)=C(1)+O(m-2)$$

recuperando de novo a variável,

$$C(N)=C(1)+O(\log_2(N)-2)$$

$$C(N)=C(1)+O(\log_2(N))-O(2)$$

$$C(N)=O(\log_2(N))$$

Exercício 4.2:

$$C(n)=2*C(\frac{n}{2})+n*\log(n)$$

$$C(n)=2*C(\frac{n}{2})+O(n*\log(n))$$

Aplicando a mudança de variável:

$$m=\log(n)\Leftrightarrow 2^m=n$$

$$C(2^m)=2*C(2^{(m-1)})+O(2^m*m)$$

$$C(2^m)=2*C(2^{(m-2)})+O(2^{m-1}*(m-1))+O(2^m*m)$$

$$C(2^m)=2*C(2^{(m-3)})+O(2^{m-2}*(m-2))+O(2^{m-1}*(m-1))+O(2^m*m)$$

…

$$C(2^m)=2*C(2^{(m-m)})+\sum_{1}^{m}O(i*2^i)$$

$$C(2^m)=2*C(1)+O(2.\frac{2^{m-1}-1}{m-1})$$

$$C(2^m) = 2 * C(1) + O(\frac{2^m - 1}{m - 1})$$

recuperando a variável:

$$C(n) = 2 * C(1) + O(\frac{n - 1}{\log(n) - 1})$$

$$C(n) = O(\frac{n - 1}{\log(n) - 1})$$