Universidade Federal do Rio Grande do Norte

Instituto Metrópole Digital

Estruturas de Dados Básicas I • IMD0029

– 1ª Avaliação, Expectativa de respostas e critério de correção –

1. [2.5 pts] Escreva uma função em C/C++ ou algoritmo em pseudocódigo que recebe um arranjo A de inteiros distintos em *ordem crescente* e retorna um índice i tal que $A[i] = i$ ou indique que tal índice não existe retornando -1. Por exemplo, se aplicado ao vetor abaixo o algoritmo deve retornar $i = 2$. Não serão consideradas soluções com complexidade temporal linear ou superior.

A:	-12	-8	2	17	28	34	35	46	57	61
	0	1	2	3	4	5	6	7	8	9

Uma vez que o vetor contém elementos distintos e ordenados, para qualquer $i > 0$, $A[i] \geq A[i-1]+1$. Portanto se criarmos um vetor B tal que $B[i] \leftarrow A[i] - i$, este vetor também estará ordenado. Neste caso, basta aplicar a busca binária sobre B procurando o elemento 0 (zero), ou seja, o elemento em que $A[i] = i$.

Conferir código fonte com resposta.

```cpp
template<std::size_t SIZE>
int searchIndexValueEqual( const std::array<int, SIZE> &A )
{
    int l = 0;
    int r = A.size() - 1;
    while ( l <= r ) {
        int m = (l+r)/2;
        int val = A[ m ] - m; // Array B[i] = A[i] - i
        if ( val == 0 ) { // Found?
            return m;
        } else if ( val > 0 ) { // Look into left side.
            r = m - 1; // Reduce right limit.
        } else {
            l = m + 1; // Reduce left limit.
        }
    }

    return -1;
}
```

Critério:

As respostas foram classificadas em três grupos:

A) 2.5 pontos:
A resposta contém o código correto baseado em busca binária.

B) 2.0 pontos:
A resposta contém o código quase correto baseado em busca binária. Porém comete um erro como, por exemplo, (1) calcular errado o índice do elemento do meio, OU (2) definir errado os limites do arranjo para a próxima fase da busca.

C) 1.0 pontos:
A resposta contém o código quase correto baseado em busca binária. Porém comete erros como, por exemplo, (1) calcular errado o índice do elemento do meio, E (2) definir errado os limites do arranjo para a próxima fase da busca.

D) Zero:
A resposta é inexistente ou totalmente inaceitável. No segundo caso, figuram as respostas cometeram 2 ou mais erros; ou apresentaram uma solução de busca linear; ou misturam busca binária com busca linear.

2. [3.0 pts] Escreva uma função em C/C++ ou algoritmo em pseudocódigo que recebe um vetor $A[0,\ldots,n-1]$ de inteiros e um índice $i \in [0, n-1]$, e rearranja os elementos de A de tal maneira que os elementos menores que $A[i]$ aparecem no início do vetor (em qualquer ordem), seguido dos elementos iguais a $A[i]$, seguido dos elementos maiores que $A[i]$ (em qualquer ordem). Por exemplo, se o vetor fornecido for $[-5, 7, 10, 7, 8, 9, 1, 7, -2, 3\ ]$ com $i = 3$, uma possível saída seria $[\ -5, 3, -2, 1, 7, 7, 7, 9, 8, 10\ ]$.

Seu algoritmo deve ter complexidade temporal *linear* e complexidade espacial *constante*. **Prove a corretude** do seu algoritmo por invariante de laço. Solução correta mas com complexidade temporal superior a linear receberá, no máximo, 30% dos pontos da questão.

Este problema é baseado no conceito de ponteiros lentos e um rápido. A classificação dos elementos acontece em 3 grupos: menores, iguais e maiores. A partição deve acontecer no próprio vetor, por meio de operações de troca. Vamos usar 4 índices, `smaller` (lento), `equal` (rápido), `larger` (lento), sendo que o último começa do final e vai avançando para o começo do vetor.

O invariante (IL) que precisamos manter a cada iteração do laço são as regiões com as seguintes caracaterísticas:

- ⋆ *Região **início**: elementos menores que ficam no subvetor $A[0 : smaller - 1]$;*
- ⋆ *Região **metade**: elementos iguais que ficam no subvetor $A[smaller : equal - 1]$;*
- ⋆ *Região **não classificados**: elementos ainda não classificados que ficam no subvetor $A[equal : larger]$. Esta região diminui a cada iteração do laço até chegar a zero;*
- ⋆ *Região **fim**: elementos maiores que ficam no subvetor $A[larger + 1 : n - 1]$, onde n é o tamanho do vetor.*

A cada iteração a região dos não classificados *é diminuída de 1 elemento, que necessariamente deve ser classificado em uma das outras 3 regiões. A prova de corretude vem do fato que o IL permanece verdadeiro antes do início do laço e durante cada iteração do laço e após o laço acabar. O laço é executado um número finito de vezes, visto que a região dos* não classificados *diminui até chegar em zero, já que $D(X) = equal - larger$.*

Conferir código fonte com resposta.

```
void partition( int A[], int l, int r, int idxPivot )
{
    auto pivot( A[ idxPivot ] ); // Storing the chosen pivot.
    auto N( r-l+1 ); // Array's size.
    /**
     * Keep the following invariants during partitioning
     * bottom group: A[0 : smaller-1]
     * middle group: A[smaller : equal-1]
     * unclassified: A[equal : larger]
     * top group: A[larger+1 : A.size()-1]
     */

    auto smaller(0), equal(0), larger(N-1);
    // Processes while there is unclassified elements.
    while( equal <= larger )
    {
        // Note that A[equal] is the incoming unclassified element.
        if ( A[equal] < pivot )
            std::swap( A[smaller++], A[equal++] );
        else if ( A[equal] == pivot )
            ++equal;
        else // A[equal] > pivot
            std::swap( A[equal], A[larger--] );
    }

    return;
}
```

O algoritmo correto vale 2 pontos, enquanto que a corretude correta vale 1 ponto.

Critério:

Nas questões de múltipla escolha, uma certa anula uma errada. Cada questão certa vale o total da questão dividido pela quantidade total de questões certas.

3. [0.8 pts] Sobre o algoritmo *intercala* ou *merge* do *mergesort*, indique a(s) opção(ões) verdadeira(s).

   (a) Precisa de memória extra para realizar a intercalação.
   (b) Combina dois vetores em um único vetor ordenado.
   (c) Tem complexidade $\Theta(log_2 n)$.
   (d) É responsável pela parte linear da complexidade do *mergesort*.
   (e) Intercala os menores elementos com os maiores.

4. [0.8 pts] Sobre o algoritmo *selection sort*, indique a(s) opção(ões) verdadeira(s):

   (a) É linear se o vetor está em ordem decrescente.
   (b) É linear se o vetor está em ordem crescente.
   (c) Realiza muitas trocas no pior caso.
   (d) Realiza poucas trocas no pior caso.
   (e) Realiza a mesma quantidade de trocas, independente do caso.
   (f) Insere elementos da parte não ordenada no parte ordenada do vetor.
   (g) Insere sempre o menor elemento da parte não ordenada na parte ordenada do vetor.

5. [0.8 pts] Qual é a complexidade temporal de pior caso do algoritmo *insertion sort* quanto utilizamos a busca binária para determinar a posição de inserção do elemento na parte ordenada do vetor?

   a) $O(n \log_2 n)$ b) $O(n \log_2 n^2)$ c) $O(n^2 \log_2 n)$ d) $O(n)$ e) $O(n^2)$

   O uso da busca binária reduz a quantiade de comparações para $O(log(n))$, contudo, o deslocamento dos elementos do vetor para a inserção é que provoca a complexidade $O(n^2)$.

   Veja, para o i-ésimo elemento faríamos aproximadamente $\log(i)$ comparações e aproximadamente j deslocamentos, o que signfica que essa operação é $O(\log(i) + i)$.

   Quando fazemos essa soma para todos os n elementos, temos:

$$\sum_{i=0}^{n}(i + \log(i)) = \frac{n(n+1)}{2} + \log(n!) = O(n^2 + n\log(n)) = O(n^2).$$

6. [1.0 pts] Em uma competição, quatro diferentes funções foram implementadas. Todas as funções usam um único laço e dentro do laço o mesmo conjunto de comandos são executados. Se $n > 0$ corresponde ao tamanho da entrada, assumindo que os comandos internos do laço não alteram o controle do laço, indique a complexidade temporal para cada item.

   (a) `for( i=0 ; i < n ; ++i )` $O(n)$ (c) `for( i=1 ; i < n ; i *= 2 )` $O(\log_2(n))$
   (b) `for( i=0 ; i < n ; i+= 2 )` $O(n)$ (d) `for( i=n ; i > 1 ; i /= 2 )` $O(\log_2(n))$

7. [1.0 pts] Com relação a características de *instabilidade*, qual(is) afirmação(ões) é(são) correta(s)? Considere as versões apresentadas em sala de aula.

   (a) Bubble sort é instável, porque quando a "bolha" maior sobe, elementos iguais podem ser trocados.
   (b) Insertion sort é instável, porque não temos controle como elementos iguais serão inseridos na parte ordenada do vetor.
   (c) Insertion sort é estável, visto que elementos iguais são inseridos na mesma ordem na parte ordenada.
   (d) Selection sort é instável, porque a cada iteração o menor elemento da vez é trocado com o primeiro elemento da parte não ordenada.

(e) Selection sort é estável, porque sempre inserimos o menor elemento na parte ordenada, preservando a ordem relativa dos elementos iguais.

(f) Merge sort é instável, porque ele não é *in-place*, ou seja, ele ordena em um vetor externo.

(g) Quick sort é estável, visto que o partição não troca elementos iguais de lugar, mas apenas elementos menores ou maiores que o pivô.

8. [1.6 pt] Responda cada uma das questões com uma breve e convincente justificativa:

(a) Se provarmos que um algoritmo possui complexidade $O(n^2)$ no pior caso, é possível que o algoritmo seja $O(n)$ para alguma entrada? *Sim, pois a complexidade quadrática indicada é a de pior caso. Pode ser que o mesmo algoritmo no melhor caso seja, por exemplo, linear.*

(b) Se provarmos que um algoritmo possui complexidade $O(n^2)$ no pior caso, é possível que o algoritmo seja $O(n)$ para todas as entradas? *Não, pois foi afirmado que a complexidade é quadrática no pior caso, então não é possível que todas as entradas sejam resolvidas em tempo linear.*

(c) Se provarmos que um algoritmo possui complexidade $\Theta(n^2)$, é possível que o algoritmo seja $O(n)$ para alguma entrada? *Não, pois a notação Θ implicam em um limite assintótico justo, ou seja, tanto o melhor quanto o pior caso, ambos apresentam complexidade quadrática.*

(d) Se provarmos que um problema possui complexidade $\Omega(n^2)$, é possível que um algoritmo que resolva tal problema seja $O(n \log_2 n)$ para alguma entrada? *Não, pois o limite inferior indicado é para o problema e não para o algoritmo. Portanto, não é possível que um algoritmo consiga resolver qualquer instância com complexidade inferior ao limite inferior estabelecido para o problema.*

(e) Se provarmos que um algoritmo possui complexidade $\Omega(n^2)$, é possível que o algoritmo seja $O(n)$ para alguma entrada? *Não, pois foi provado que o limite inferior, ou seja, o melhor tempo para o melhor caso é quadrático. Portanto, não é possível que o mesmo algoritmo resolva qualquer instância em tempo linear.*

(f) Podemos afirmar que $f(n) = \Theta(n^2)$, sendo que $f(n) = 100n^2$ quando n é par e $f(n) = 20n^2 - n \log_2 n$ quando n é impar? *Sim, pois independente do valor de n, a função que determina a complexidade tanto no pior caso quanto no melhor caso é quadrática. Portanto é corretor afimar que o algoritmo possui complexidade Θ (justa) quadrática.*

(g) Se um problema possui complexidade $\Omega(n)$, é possível que um algoritmo que resolve o problema possua complexidade $\Theta(n \log_2 n)$? *Sim, pois tal complexidade não contraria o limite inferior do problema.*

(h) Se um algoritmo é considerado ótimo para resolver um problema de complexidade cúbica, este algoritmo pode ter complexidade no pior caso $O(n^2)$? *Não, pois se o algoritmo é considerado ótimo, implica afirmar que a complexidade de seu pior caso (no caso n^3) corresponde a complexidade mínima para resolver o problema. Desta forma, não é possível afirmar que o mesmo algoritmo consiga resolver qualquer entrada com complexidade quadrática.*

$\sim$ FIM $\sim$