Universidade Federal do Rio Grande do Norte

Instituto Metrópole Digital

Estruturas de Dados Básicas I • IMD0029

– 1ª Avaliação, 6 de abril de 2017 **Expectativa de resposta** –

1. [2.5 pts] Um algoritmo óbvio para calcular x^n usa $n-1$ multiplicações. Um algoritmo recursivo pode ser melhor. Quanto $n \leq 1$ temos o caso base. Caso contrário, se n for *par*, temos que $x^n = x^{n/2} \cdot x^{n/2}$, e se n for *ímpar*, $x^n = x^{(n-1)/2} \cdot x^{(n-1)/2} \cdot x$.

   Por exemplo, para calcular x^{62}, são feitas os seguintes cálculos (9 multiplicações):

   $$x^{62} = (x^{31})^2;\;\; x^{31} = (x^{15})^2 \cdot x;\;\; x^{15} = (x^7)^2 \cdot x;\;\; x^7 = (x^3)^2 \cdot x;\;\; x^3 = (x^2)^2 \cdot x$$

   já o algoritmo óbvio precisaria de 61 multiplicações.

   Escreva uma função em C/C++ ou algoritmo em pseudocódigo **recursivo** que implementa esta ideia, ou seja, você precisa implementar uma função `pow(a,b)` que retorna a^b.

   Para o algoritmo dar certo, temos como pré-requisito $n \geq 0$.

```
unsigned long long
power( long a, long n )
{
    if ( n == 0 ) return 1;
    if ( n == 1 ) return a;
    auto x = power( a, n/2 ); // or power( a, std::floor( n/2 ) )

    if ( n % 2 == 0 )  return x * x;
    else               return a * x * x;
}
```

   ou

```
unsigned long long
power2( long a, long n )
{
    if ( n == 0 ) return 1;
    if ( n == 1 ) return a;

    if ( n % 2 == 0 )  return pow( a*a, n/2 );
    else               return pow( a*a, n/2 ) * a;
}
```

   Sendo a multiplicação como operação dominante, temos a seguinte relação de recorrência:

   $$\text{Número total de multiplicações} = \begin{cases} M(0) & = M(1) = 0, \\ M(n) & = M(\lfloor n/2 \rfloor) + 1, \text{ se } n \text{ for par}, \\ M(n) & = M(\lfloor n/2 \rfloor) + 2, \text{ se } n \text{ for ímpar}. \end{cases}$$

   A complexidade de pior caso ocorre se todos os n intermediários forem ímpares, ou seja, sempre executamos a última equação da recorrência. Usando o método da substituição progressiva, temos:

   $$\begin{aligned} M(n) &= M(\lfloor n/2 \rfloor) + 2 & & \text{passo 1} \\ &= [M(\lfloor n/4 \rfloor) + 2] + 2 & = M(\lfloor n/4 \rfloor) + 4 & \quad \text{passo 2} \\ &= [M(\lfloor n/8 \rfloor) + 2] + 4 & = M(\lfloor n/8 \rfloor) + 6 & \quad \text{passo 3} \\ &= [M(\lfloor n/16 \rfloor) + 2] + 6 & = M(\lfloor n/16 \rfloor) + 8 & \quad \text{passo 4} \\ &= \cdots & & \\ &= M(\lfloor n/2^k \rfloor) + 2 \cdot k & & \text{passo } k \end{aligned}$$

Eventualmente chegaremos no caso base ímpar, ou seja, $\frac{n}{2^k} = 1$ logo $k = \log_2 n$. Substituindo na equação geral $M(\lfloor n/2^k \rfloor) + 2 \cdot k$, temos:

$$M(n) = M(1) + 2 \cdot \log_2 n = 0 + 2 \cdot \log_2 n = 2 \cdot \log_2 n.$$

Este é o limite (superior) do pior caso, visto que o número de multiplicações no caso em que n e suas divisões são sempre ímpar é maior do que se for par.

No melhor caso, $n = 0$ ou $n = 1$, temos zero multiplicações.

Critério de correção:

O algoritmo solicitado se totalmente correto vale 2.5 pontos.

Pequenos erros no algoritmo, como por exemplo esquecer casos especiais ou não usar as multiplicações corretamente, perde 0.5 ponto **por erro**.

Chamadas recursivas desnecessárias reduz 0.5 ponto.

Falha em retornar o valor calculado na recursão, reduz 0.5 ponto.

Misturar iteração com recurção perde 1.0 ponto.

2. [3.0 pts] Um vetor A de comprimento n é considerado **ordenado ciclicamente** se o menor elemento do vetor está no índice i, e a sequência $\langle A[i], A[i+1], \ldots, A[n-1], A[0], A[1], ..., A[i-1] \rangle$ está ordenada em *ordem crescente*, como ilustrado abaixo (menor elemento é 37, índice $i = 4$).

A:	78	85	92	93	37	42	51	58	64	67
	0	1	2	3	4	5	6	7	8	9

Escreva uma função em C/C++ ou algoritmo em pseudocódigo com complexidade $O(\log_2 n)$ que encontra e retorna a posição do menor elemento em um vetor ciclicamente ordenado. Assuma que todos os elementos são distintos. Para o exemplo acima, sua função deveria retornar a posição 4.

Para aderir às restrições de complexidade temporal do enunciado, $O(\log n)$, aplicamos o paradigma de divisão e conquista, tomando como base uma estratégia parecida com a de busca binária.

O algoritmo mantém um intervalo de elementos candidatos e, iterativamente, elimina aproximadamente metade da faixa de candidatos. Seja $I = [l, r]$ o conjunto de índices sob consideração, m o elemento localizado no meio de I, i.e. $l + \lfloor \frac{r-l}{2} \rfloor$. Como não pode haver repetição de elementos, temos apenas 2 possibilidades ao comparar dois elementos quaisquer, digamos a e b, do intervalo I: ou $a > b$ ou $a < b$. Considerando o elemento do meio, $A[m]$ e o elemento da extrema direita de I, $A[r]$, temos duas possibilidades para considerar[1]*:*

(a) *Se $A[m] > A[r]$ então sabemos que o menor elemento está na segunda metade e, consequentemente, seu índice não pode estar no intervalo $[l, m]$. Desta forma, a busca deve ser realizada no intervalo $[m+1, r]$. Note que $A[m]$* **é excluído do novo intervalo** *visto que claramente ele nao pode ser o menor, já que, pelo menos, é maior que $A[r]$.*

(b) *Se $A[m] < A[r]$, o candidato está na metade esquerda, ou $[l, m]$. Ao contrário da opção anterior,* **o elemento do meio $A[m]$ ainda é candidato a ser o menor** *do intervalo esquerdo, visto que nada se pode afirmar sobre sua relação com os elementos à sua esquerda.*

O algoritmo inicia com o intervalo completo, ou seja, $[first, last)$. A cada iteração o intervalo de candidatos é reduzido pela (aproximadamente) metade. O algoritmo finaliza com um intervalo I que contém apenas 1 elemento: o menor de todos.

[1] Um raciocínio similar pode ser estabelecido se considerarmos a relação do elemento do meio, $A[m]$, com o elemento na extrema esquerda, ou $A[l]$.

```
int * smallest_in_cyclic_sorted( int * first_, int * last_ )
{
    int l = 0, r = std::distance(first_, last_)-1;
    while ( l < r ) // Enquanto houver pelo menos 2 elementos no intervalo
    {
        int m = l + ( (r-l)/2 ); // Ache o meio
        if ( first_[m] > first_[r] )  l = m + 1; // va para direita.
        else                          r = m; // va para esquerda
    }

    return first_+l; // 'l' aponta para o menor.
}
```

Critério de correção:

As respostas foram classificadas em cinco grupos:

A) 3.0 pontos:
A resposta contém o código correto, com complexidade $O(\log n)$.

B) 2.5 pontos:
A resposta é quase correta, mas com pequenos erros que para casos específicos, mas no geral o algoritmo funciona.

C) 2.0 pontos:
A resposta apresenta 1 tipo de erro abaixo:

- ⋆ falha para determinar que metade explorar OU
- ⋆ erros na definição dos limites da metade OU
- ⋆ falha na condição de parada do laço.

D) 1.0 ponto:
Algoritmo correto mas com complexidade linear. Algoritmos com 2 ou mais erros descritos na categoria anterior ou equivalentes.

E) Zero:
A resposta é inexistente ou totalmente inaceitável.

3. [2.5 pts] Escreva uma função em C/C++ ou algoritmo em pseudocódigo que **não use mais do que** $\lceil 3n/2 \rceil - 2$ **comparações** para encontrar e retornar os elementos *máximo* e *mínimo* em um vetor com $n > 0$ inteiros. Descreva como você contou as comparações da sua solução.

Desejamos achar o menor e maior elementos, o par (m, M) respectivamente, em um intervalo.

Se $n = 1$, não precisamos de comparação alguma: $m = M =$ elemento único do intervalo.

Se $n > 1$, então basta 1 comparação para determinar o maior e menor entre os dois primeiros. Ou seja, temos que $(m, M) =$ `minmax(*first,*(first+1))`, onde `minmax` é uma função que retorna um par contendo o menor e maior elementos entre os dois elementos passados como argumento. Esta função, obviamente, gasta exatamente 1 comparação.

A partir daí, processamos os elementos restantes, **dois a dois**, *atualizando o par de candidatos a mínimo/máximo (m, M) quando necessário, da seguinte forma: Seja (x, y) um dos pares de elementos restantes. Se $min(x, y) < m$, atualizamos m; se $max(x, y) > M$ atualizamos M. No total temos 3 comparações a cada par processado:*

(1) 1 comparação da chamada de `minmax` sobre o par (x, y) para determinar o maior e menor entre os dois;

(2) 1 comparação para determinar o menor entre o menor do passo (1) e m (o menor até o momento); e,

(3) 1 comparação para determinar o maior entre o maior do passo (1) e M (o maior até o momento).

Portanto, para $n > 1$ temos:

- ⋆ *1 comparação para achar o par (m, M) inicial;*
- ⋆ **Se n for par**: *$(n-2)/2 \times 3$ comparações para os demais pares até o final do intervalo.*
- ⋆ **Se n for ímpar**:*$(n-3)/2 \times 3$ comparações para os demais pares até o final do intervalo $+\ 2$ comparações no final entre (m, M) e o último elemento.*

Assim temos a seguinte quantidade de comparações:

$$comparisons(n) = \begin{cases} 1 + \frac{n-2}{2} \times 3 & = 1 + \frac{3n}{2} - \frac{6}{2} & = \frac{3n}{2} - 2, & \text{se } n \text{ é par.} \\ 1 + \frac{n-3}{2} \times 3 + 2 & = 1 + \frac{3n}{2} - \frac{9}{2} + 2 & = \frac{3n}{2} - 1.5 & \text{se } n \text{ é ímpar.} \end{cases}$$

Portanto, condensando os dois casos temos que o número de comparações é $\lceil \frac{3n}{2} \rceil - 2$ ■.

```
std::pair<int,int> min_max( const int *first_, const int *last_ )
{
    auto n = std::distance( first_, last_ ); // # of elements in range.

    if ( n == 0 ) // unexpected case.
        throw std::invalid_argument( "Range must have at least 1 element!" );
    if ( n == 1 ) // simple case: min = max.
        return std::make_pair( *first_, *first_ );
    // Two or more elements in range: prepare the initial candidates as a pair (min, max)
    std::pair<int,int> minmax_pair = std::minmax( *first_, *( first_+ 1 ) );

    // Visit the rest of the range, two elements at a time.
    for ( auto i(2) ; i+1 < n ; i+=2 ) {
        // Get the min/max for the next two elements.
        auto candidate_pair = std::minmax( *(first_+i), *(first_+i+1) );

        // Compare the candidate pair against the current min & current max.
        minmax_pair = { std::min( candidate_pair.first,  minmax_pair.first ),
                        std::max( candidate_pair.second, minmax_pair.second ) };
    }

    if ( n % 2 == 1 ) // Case: range has odd number of elements.
    {   // Compare the last element agains the current min/max pair.
        minmax_pair = { std::min( *(first_+ (n-1) ) , minmax_pair.first  ),
                        std::max( *(first_+ (n-1) ) , minmax_pair.second ) };
    }

    return minmax_pair;
}
```

Critério de correção:

O algoritmo vale 1.5 pontos e correta descrição da contagem de comparações vale 1 ponto.

As respostas foram classificadas em cinco grupos:

A) 2.5 pontos:
A resposta contém o código correto, com complexidade linear $\lceil 3n/2 \rceil - 2$ e uma correta descrição da contagem de comparações.

B) 2.0 pontos:
A resposta contém o código correto, com complexidade linear $\lceil 3n/2 \rceil - 2$, mas a descrição de contagem de comparações não é adequada.

C) 1.0 ponto:
Algoritmo correto, mas com complexidade $2n - 3$, com a descrição de comparações correta.

D) 0.5 ponto:
Algoritmo incorreto (por exemplo, não retorna nada para o cliente), ou correto $(2n - 3)$ mas sem descrição de comparações.

E) Zero:
A resposta é inexistente ou totalmente inaceitável.

Nas questões 4 a 21 marque a(s) opção(ões) que você considera correta. Cada questão vale 0.25 pt.

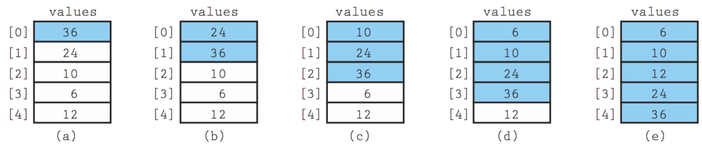
Critério de correção:

Cada questão com todas as opções corretas indicadas ganha 0.25.

Se a quantidade de opções corretas $>$ opções erradas, ganha-se metade dos pontos da questão.

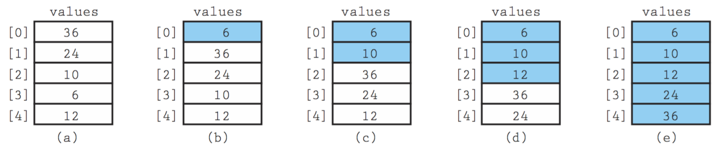
Senão se a quantidade de opções corretas $<=$ opções erradas, ganha-se zero pontos da questão.

4. Qual é o algoritmo de ordenação representado na imagem abaixo



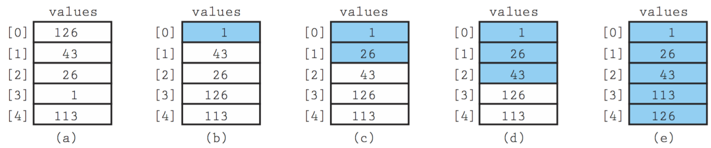
a) √ insertion b) bubble c) selection d) merge

5. Qual é o algoritmo de ordenação representado na imagem abaixo



a) insertion b) √ bubble c) selection d) merge

6. Qual é o algoritmo de ordenação representado na imagem abaixo



a) insertion b) bubble c) √ selection d) merge

7. De que maneira a busca binária (BB) poderia ser útil para o selection sort?

   (a) Para ordenar o arranjo antes de realizar a busca.
   (b) √ De maneira alguma, são algoritmos com propósitos diferentes.
   (c) Podemos usar a BB para achar o menor elemento para troca.
   (d) Usamos a mesma estratégia da BB no selecion sort, mas ao invés de buscar fazemos a ordenação.

8. De que maneira a busca binária poderia ser útil para o insertion sort?

   (a) De maneira alguma, são algoritmos com propósitos diferentes.
   (b) Para localizar o menor elemento na parte ainda não processada.
   (c) √ Para achar o local certo de inserção do item sendo processado.
   (d) Para localizar o menor elemento na parte ordenada.

9. A complexidade temporal da busca linear quando um elemento não está na lista é:

   (a) √ Proporcional ao tamanho da lista.
   (b) Constante, independente da organização dos dados.
   (c) Proporcional ao tamanho da metade da lista.
   (d) Independente da organização dos dados.
   (e) √ Constante, se a lista está ordenada.

10. Quantas comparações são executadas se aplicarmos o selection sort em um vetor de 100 elementos já ordenados?

    a) 9900 b) 10000 c) $\surd$ 4950 d) 99 e) 9801

    $T_{\text{selection}}(n) = \frac{n(n-1)}{2} = \frac{(100*99)}{2} = 4950$

11. Em linhas gerais, a busca binária por uma chave K segue a estratégia de:

    (a) Dividir a lista em 2 metades e procurar K nas metades.
    (b) Dividir o vetor em elementos ordenados e não ordenados.
    (c) Ordenar a lista, para aplicar a busca linear.
    (d) Particionar a lista em elementos $\leq K$ e $> K$.
    (e) $\surd$ Se elemento central não for K, buscar em uma das metades.

12. Merge sort (MS) é usado para ordenar 1000 elementos em um vetor. Qual afirmação relativa a complexidade temporal é verdadeira?

    (a) MS é mais eficiente se o vetor está em ordem não decrescente.
    (b) MS é menos eficiente se o vetor está em ordem aleatória.
    (c) MS é mais eficiente se o vetor está em ordem não crescente.
    (d) $\surd$ Em termos de eficiência, tanto faz a organização do vetor.

13. Suponha um vetor em ordem não decrescente. Qual algoritmo levará mais tempo para executar?

    a) $\surd$ Quick sort, com o 1º elemento como pivô. b) Bubble sort.
    c) Insertion sort. d) Merge sort.

14. Com relação a características de **instabilidade**, quais afirmações são corretas? 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) $\surd$ Insertion sort é estável, visto que elementos iguais são inseridos na mesma ordem na parte ordenada.
    (d) $\surd$ 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.

15. O quick sort é um algoritmo **ótimo** pois sua complexidade é $O(n \log_2 n)$.

    (a) Falso, sua complexidade é $\Theta(n \log_2 n)$.
    (b) $\surd$ Falso, sua complexidade de pior caso é $O(n^2)$.
    (c) Verdadeiro, é eficiente em todos os casos.
    (d) Falso, pois sua complexidade no melhor caso é $O(n)$.

16. O selection sort é recomendado se os elementos ordenados têm grande *footprint* de memória.

    (a) $\surd$ Verdadeiro, pois ele realiza poucas trocas.

(b) Falso, pois o elemento não vai logo para a posição final.
(c) Verdadeiro, pois ele realiza poucas comparações.
(d) Falso, visto que ele é $\Theta(n^2)$ independente da organização.

17. Seja k um índice válido em um vetor qualquer de n elementos. Qual a complexidade para acessar k?

a) $O(k)$ b) $\surd$ $O(1)$ c) $O(n)$ d) $O(\log_2 n)$ e) $O(n^2)$ f) $O(\log_{10} n)$

18. O selection sort no pior caso é melhor do que o insertion sort no pior caso.

a) Sim. b) Não. c) $\surd$ São iguais. d) Depende dos dados.

19. Durante a execução do quick sort o vetor é recursivamente dividido em partes iguais.

(a) Não. A divisão igual só ocorre em vetores de tamanho par.
(b) Sim. Ele e o merge sort adotam a estratégia divisão-e-conquista.
(c) $\surd$ Não. O tamanho das partes depende do pivô.
(d) Sim. Por isso sua complexidade é $O(n \log_2 n)$.

20. O algoritmo *intercala* ou *merge* do merge sort...

(a) Tem complexidade $O(log_2 n)$.
(b) $\surd$ Passa apenas uma vez em cada vetor para gerar um vetor ordenado.
(c) Recebe dois vetores e os retorna ordenados.
(d) Intercala os menores elementos com os maiores.

21. Sobre o algoritmo partição do quick sort...

(a) É responsável pela parte $O(\log_2 n)$ da complexidade do quick sort.
(b) Compacta os elementos menores que o pivô no início.
(c) Posiciona o elemento central em seu local definitivo.
(d) $\surd$ Faz o quick sort ser quadrático se uma das metades é sempre vazia.

$\sim$ FIM $\sim$