Questão **1**

Resposta salva

Vale 1,00 ponto(s).

Marcar questão

Assinale a afirmação **correta** a respeito dos algoritmos de ordenação estudados:

- a. Supondo um cenário de execução do Quicksort, em que todas as chamadas ao particiona gera um particionamento desbalanceado, sempre colocando 1/10 dos elementos em uma das partições e 9/10 dos elementos na outra partição, a função T que descreve o tempo de execução neste cenário é descrita pela seguinte recorrência: T(n) = T(n/10) + T(9n/10) + Theta(n), o que ainda resulta em uma complexidade de tempo igual a Theta(n*log2(n)).
- b. O Heapsort pode ser considerado uma versão melhorada do Selection sort, no qual a seleção do próximo valor a ser colocado na porção ordenada do vetor é feita de modo muito mais eficiente (complexidade linear) do que a varredura sequencial (complexidade logarítmica) feita no Selection sort.
- c. A função T que determina o tempo de execução do Merge sort é expressa pela seguinte recorrência: T(n) = 2T(n - 2) + Theta(n).
- d. A função T que determina o tempo de execução do Quicksort, no cenário de pior caso, é expressa pela seguinte recorrência: T(n) = 2T(n/2) + Theta(n).
- e. A função que determina o tempo de execução do Quicksort, no cenário de melhor caso, é expressa por uma recorrência diferente daquela do algoritmo Merge sort.

Limpar minha escolha

Questão **2**

Resposta salva

Vale 1,00 ponto(s).

Marcar questão

Considere a seguinte variação do algoritmo de ordenação *merge sort*:

```
void merge_sort(int * a, int ini, int fim){
  if(ini < fim){
    int q1 = ini + (fim - ini) / 4;
    merge_sort(a, ini, q1);
    merge_sort(a, q1 + 1, fim);
    merge(a, ini, q1, fim);
  }
}
```

Ao comparar essa variação com a versão clássica do algoritmo, é incorreto afirmar que (pode haver mais de uma afirmação incorreta):

- a. Essa variação difere da versão clássica do *merge sort* ao adotar uma nova estratégia de divisão do problema em subproblemas menores. Nesta variação do algoritmo, um vetor de tamanho **n** é dividido em subvetores de tamanho **n/4** e **3n/4**, que são ordenados recursivamente, e depois combinados com a função *merge*.
- b. A nova estratégia de divisão do problema em subproblemas não muda a complexidade assintótica da função *merge* (que continua linear), porém piora a complexidade do algoritmo *merge sort*, por apresentar uma árvore de recursão mais profunda que a observada na versão clássica.
- c. A nova estratégia para divisão do problema em subproblemas não muda a complexidade assintótica nem da função *merge* (o fato de combinar dois subvetores de tamanhos distintos não é relevante, já que o que importa é a quantidade total de elementos combinados), e nem do algoritmo *merge sort* (apesar de a árvore de recursão ser mais profunda quando comparada à árvore da versão clássica, sua altura continua sendo logarítmica).
- d. A nova estratégia de divisão do problema em subproblemas piora tanto a complexidade assintótica da função *merge* (é necessário realizar mais comparações para combinar os dois subvetores ordenados), quanto do algoritmo *merge sort* por conta da árvore de recursão mais profunda em comparação com a árvore de recursão da versão clássica.
- e. Com a nova estratégia de divisão do problema em subproblemas, ao combinar um subvetor de tamanho **n/4** com outro de tamanho **3n/4**, a função *merge* passará a realizar, no cenário de **melhor caso**, apenas **n/4** comparações (contrastando com as **n/2** comparações necessárias, no cenário de melhor caso, para combinar dois subvetores de tamanho **n/2** na versão clássica do algoritmo).

Questão **3**

Resposta salva

Vale 1,00 ponto(s).

Marcar questão

Considere uma versão alternativa do algoritmo de ordenação *quicksort* que utiliza a versão da função *particiona* apresentada a seguir:

```
int particiona(int * a, int ini, int fim){
  int i, min, max, x, prox, q, tam_b;
  int * b;

  tam_b = fim - ini + 1;
  b = calloc(tam_b, sizeof(int));
  min = a[ini];
  max = a[ini];

  for(i = ini + 1; i <= fim; i++){

    if(a[i] < min) min = a[i];
    if(a[i] > max) max = a[i];
  }

  x = (min + max) / 2;
  prox = 0;

  for(i = ini; i <= fim; i++){

    if(a[i] <= x){
      b[prox] = a[i]; prox++;
    }
  }

  q = ini + prox - 1;

  for(i = ini; i <= fim; i++){

    if(a[i] > x){
      b[prox] = a[i]; prox++;
    }
  }

  for(i = 0; i < tam_b; i++){
    a[ini + i] = b[i];
  }

  return q;
}
```

A partir do código, e assumindo que todo vetor a ser ordenado não possui valores repetidos, assinale a alternativa correta:

- a. O algoritmo deixará de funcionar corretamente com essa versão da função *particiona*.
- b. O algoritmo continuará funcionando corretamente com essa versão do *particiona*, mas ela não garante particionamentos balanceados, o que pode levar o algoritmo a apresentar uma complexidade assintótica Theta(n^2).
- c. O algoritmo não só continuará funcionando corretamente com essa versão do *particiona*, como irá prevenir particionamentos desbalanceados que poderiam levar o algoritmo a apresentar uma complexidade assintótica Theta(n^2), uma vez que usa como valor pivô a média entre o maior e o menor valor de um (sub)vetor.
- d. A versão alternativa da função *particiona* é assintóticamente pior do que a versão clássica. Tal piora se deve aos 4 loops '*for*' que são executados na função.
- e. Nenhuma das demais afirmações está correta.

Limpar minha escolha

Questão **4**

Resposta salva

Vale 1,00 ponto(s).

Marcar questão

Considere um vetor de valores inteiros, organizado da seguinte forma:

$\{ a_1, b_1, c_1, a_2, b_2, c_2, ..., a_m, b_m, c_m \}$

onde:

$a_1 > b_1 > c_1$ e $c_{i+1} > a_1$.

Dado que o vetor possui um total de **n** valores (observe que **n = 3m**), assinale a alternativa incorreta:

- a. Se tal vetor for ordenado com o *bubble sort*, serão feita ao todo **n trocas** de elementos no vetor em duas varreduras do algoritmo, sendo **2m** trocas na primeira varredura, e **m** trocas na segunda.
- b. Se tal vetor for ordenado com o *merge sort*, este algoritmo rodará em tempo **Theta(nlgn)**, pois esse algoritmo não é sensível à forma como os **n** elementos estão organizados previamente dentro do vetor.
- c. Assuma uma versão da função *particiona* com o seguinte funcionamento: escolhe o primeiro elemento de um (sub)vetor como pivô, e coloca na partição da esquerda todos os valores menores ou iguais ao pivô quando há pelo menos 1 elemento estritamente maior que o pivô no (sub)vetor; caso todos os elementos sejam menores ou iguais ao pivô, coloca na partição da esquerda apenas os elementos estritamente menores que o pivô. Se tal vetor for ordenado com o *quick sort*, este algoritmo rodará em tempo **Theta(nlgn)**.
- d. Se tal vetor for ordenado com o *selection sort*, este algoritmo rodará em tempo **quadrático**, uma vez que a varredura para seleção do menor elemento do vetor (ou de uma porção do vetor) não é beneficiada pelo esquema de organização apresentado, e continua sendo executada com complexidade Theta(n).
- e. Se tal vetor for ordenado com o *insertion sort*, este algoritmo rodará em **tempo linear**, pois o número de deslocamentos necessários para inserir cada elemento na sua posição final nunca será maior que 2.

Limpar minha escolha

Questão **5**

Resposta salva

Vale 1,00 ponto(s).

Marcar questão

Assinale as afirmações falsas (pode haver mais de uma) referentes a estruturas do tipo lista:

- [x] a. Se os elementos forem mantidos em uma lista por ordem de valor, a complexidade assintótica (no que diz respeito ao tempo de execução) da busca pode ser reduzida para **O(lgn)**, independente do tipo da lista.
- [ ] b. Quando já se sabe o endereço de um nó referente ao antecessor de um elemento a ser removido de uma lista ligada, a operação de remoção pode ser executada em **tempo constante**, uma vez que envolve apenas o acerto de ponteiros. Já em uma lista sequencial, mesmo que já se saiba o índice do elemento a ser removido, sua remoção terá custo de tempo **O(n)** por conta de eventuais deslocamentos de elementos dentro do vetor.
- [ ] c. Em listas não ordenadas, a complexidade assintótica (no que diz respeito ao tempo de execução) da busca será sempre **O(n)**.
- [x] d. A operação de inserção de um elemento **e** em um índice **i** é assintoticamente melhor (no que diz respeito ao tempo de execução) em uma lista sequencial do que em uma lista ligada.
- [ ] e. Listas ligadas usam a memória de forma mais racional, ocupando apenas a memória necessária para armazenar seu conjunto de elementos.

Questão **6**

Resposta salva

Vale 1,00 ponto(s).

Marcar questão

Considere a implementação de lista disponibilizada no arquivo **lista_sequencial.c**. Qual será o estado resultante da lista após a execução das operações abaixo?

```
Lista * lista = cria_lista(8);
insere(lista, 30, 0);
insere(lista, 20, 1);
insere(lista, 40, 1);
insere(lista, 10, 0);
insere(lista, 50, 1);
insere(lista, 70, 2);
insere(lista, 60, 3);
```

- a. 10 60 70 50 30 20 40
- b. 10 70 50 60 20 40 30
- c. 10 20 30 40 60 60 70
- d. 10 50 70 60 30 40 20
- e. 30 20 40 10 50 70 60

Limpar minha escolha

Questão **7**

Resposta salva

Vale 1,00 ponto(s).

Marcar questão

Quantos deslocamentos de elementos são realizados no total como resultado das operações realizadas na questão anterior?

- a. 12
- b. 13
- c. 14
- d. 10
- e. 11

Questão **8**

Resposta salva

Vale 1,00 ponto(s).

Marcar questão

Se ao invés de uma lista sequencial, tivessemos usado a lista ligada implementada no arquivo **lista_ligada.c** (assumindo seu uso como uma lista **não ordenada**) para realizar as operaçõs de inserção da **Questão 6**, quantos nós seriam visitados no total? Considere por "nós visitados" todos os nós pelos quais precisamos passar como consequência direta ou indireta das inserções que são realizadas (o nó novo que é criado para uma dada operação de inserção não é contabilizado como "nó visitado" pois não precisamos passar por ele para concluir a operação).

- a. 9
- b. 29
- c. 28
- d. 10
- e. 8

Limpar minha escolha

Questão **9**

Resposta salva

Vale 1,00 ponto(s).

Marcar questão

Considere uma função que recebe duas listas ligadas **l1** e **l2**, cada uma com **n** elementos, e concatena os nós da lista **l2** ao final da lista **l1**. Qual a complexidade assintótica desta função considerando a implementação de lista ligada do arquivo **lista_ligada.c**.

- a. Theta(n^2)
- b. Theta(nlg(n))
- c. Theta(1)
- d. Theta(n)
- e. Theta(lg(n))

Limpar minha escolha

Questão **10**

Resposta salva

Vale 1,00 ponto(s).

Marcar questão

Qual seria complexidade assintótica da função considerada no exercício anterior se a implementação da lista ligada for modificada para também manter um ponteiro para o último nó da lista?

- a. Theta(1)
- b. Theta(lg(n))
- c. Theta(n^2)
- d. Theta(n)
- e. Theta(nlg(n))

Limpar minha escolha

Questão **11**

Resposta salva

Vale 1,00 ponto(s).

Marcar questão

Considere uma lista sequencial que realoca o vetor (*array*) de elementos quando uma nova inserção vai ser realizada e a mesma já se encontra completamente ocupada. Se no instante em que a realocação acontece a lista possui **n** elementos armazenados, qual a complexidade assintótica desta operação de realocação?

- a. Theta(n^2)
- b. Theta(lgn)
- c. Theta(n)
- d. Theta(nlgn)
- e. Theta(1)

Limpar minha escolha

Questão **12**

Resposta salva

Vale 1,00 ponto(s).

Marcar questão

A partir do cenário proposto na questão anterior, como fica a complexidade assintótica da operação de inserção nas ocasiões em que a realocação acontece?

- a. Se mantém em O(n)
- b. Muda de O(lgn) para Theta(n)
- c. Muda de O(n) para Theta(n)
- d. Muda de O(1) para O(lgn)
- e. Muda de O(n) para O(nlgn)

Limpar minha escolha

Questão **13**

Resposta salva

Vale 1,00 ponto(s).

Marcar questão

Considerando ainda a lista sequencial com realocação da **Questão 11**: se vetor (*array*) sempre é realocado de modo a ter 100 posições a mais do que tinha antes, quantas realocações são feitas ao longo de 950 inserções de elementos em uma lista inicialmente vazia e com capacidade inicial para armazenar 100 elementos?

- a. 12
- b. 9
- c. 10
- d. 11
- e. 8

Limpar minha escolha

Questão **14**
Resposta salva
Vale 1,00 ponto(s).
Marcar questão

Considerando a política de realocação descrita na **questão anterior**: qual o desperdício máximo de memória (isto é, a quantidade de memória alocada, mas não usada efetivamente para guardar informação útil) em uma lista sequencial cujo vetor (*array*) alocado tem, em um dado instante, **k posições** (com k > 100)? Assuma que desde a criação da lista, nenhuma remoção de elemento foi realizada, apenas inserções.

a. 98
b. 100
c. 99
d. k - 1
e. k / 2

Limpar minha escolha

Questão **15**
Resposta salva
Vale 1,00 ponto(s).
Marcar questão

E se o vetor (*array*) fosse realocado de modo a ter o dobro de posições em relação ao que tinha antes, quantas realocações seriam feitas ao longo de 950 inserções de elementos em uma lista inicialmente vazia e com capacidade incial para armazenar 100 elementos?

a. 10
b. 4
c. 6
d. 12
e. 8

Limpar minha escolha

Questão **16**
Resposta salva
Vale 1,00 ponto(s).
Marcar questão

Considerando, desta vez, a política de realocação descrita na **questão anteior**: qual o desperdício máximo de memória (isto é, a quantidade de memória alocada, mas não usada efetivamente para guardar informação útil) em uma lista sequencial cujo vetor (*array*) alocado tem, em um dado instante, **k posições** (com k > 100)? Assuma que desde a criação da lista, nenhuma remoção de elemento foi realizada, apenas inserções.

a. k / 2 + 1
b. k / 2 - 1
c. k
d. k - 1
e. k / 2

Limpar minha escolha

Questão **17**
Resposta salva
Vale 1,00 ponto(s).
Marcar questão

Considere a implementação de árvore binária implementada no arquivo **arvore_binaria.c**, e o conjunto de funções para uso como uma árvore de propósito geral (ou seja, sem ordenação). Em que ordem os valores armazenados na árvore serão apresentados, em um percurso *in-ordem*, após a execução do seguinte trecho de código:

```
Arvore * arvore = cria_arvore();
insere(arvore, NULL, 100, INDEFINIDO);
insere(arvore, busca(arvore, 100), 90, ESQUERDO);
insere(arvore, busca(arvore, 100), 80, DIREITO);
insere(arvore, busca(arvore, 90), 10, ESQUERDO);
insere(arvore, busca(arvore, 90), 70, DIREITO);
insere(arvore, busca(arvore, 70), 20, ESQUERDO);
insere(arvore, busca(arvore, 70), 5, DIREITO);
insere(arvore, busca(arvore, 80), 50, ESQUERDO);
insere(arvore, busca(arvore, 80), 30, DIREITO);
imprime(arvore);
```

a. 100 90 80 10 70 50 30 20 5
b. 100 90 10 70 20 5 80 50 30
c. 5 10 20 30 50 70 80 90 100
d. 10 90 20 70 5 100 50 80 30
e. 100 90 80 10 70 20 5 50 30
f. 10 20 5 70 90 50 30 80 100

Limpar minha escolha

Questão **18**
Resposta salva
Vale 1,00 ponto(s).
Marcar questão

Considere a implementação de árvore binária implementada no arquivo **arvore_binaria.c**, mas agora o conjunto de funções para uso como uma árvore binária de busca. Qual será a altura da árvore resultante da execução do seguinte trecho de código:

```
Arvore * arvore = cria_arvore();
insere_ord(arvore, 100);
insere_ord(arvore, 90);
insere_ord(arvore, 80);
insere_ord(arvore, 10);
insere_ord(arvore, 70);
insere_ord(arvore, 20);
insere_ord(arvore, 5);
insere_ord(arvore, 50);
insere_ord(arvore, 30);
```

a. 4
b. 8
c. 3
d. 5
e. 7
f. 6

Limpar minha escolha

Questão **19**
Resposta salva
Vale 1,00 ponto(s).
Marcar questão

Considerando, novamente, a implementação de árvore binária implementada no arquivo **arvore_binaria.c** e o conjunto de funções para uso como uma árvore binária de busca. Qual será a altura da árvore resultante da execução do seguinte trecho de código:

```
Arvore * arvore = cria_arvore();
insere_ord(arvore, 80);
insere_ord(arvore, 100);
insere_ord(arvore, 50);
insere_ord(arvore, 90);
insere_ord(arvore, 10);
insere_ord(arvore, 70);
insere_ord(arvore, 20);
insere_ord(arvore, 5);
insere_ord(arvore, 30);
insere_ord(arvore, 105);
insere_ord(arvore, 120);
```

a. 8
b. 3
c. 4
d. 6
e. 5
f. 7

Limpar minha escolha

Questão **20**
Resposta salva
Vale 1,00 ponto(s).
Marcar questão

Considerando a árvore criada a partir das operações listadas na questão anterior, qual é o valor do elemento correspondente ao nó mais desbalanceado da árvore?

a. 105
b. 10
c. 80
d. 20
e. 50
f. 100

Limpar minha escolha

Questão **21**
Resposta salva
Vale 1,00 ponto(s).
Marcar questão

Considerando, ainda, a árvore resultante das operações relacionadas na **Questão 19**, assinale a alternativa que indica, corretamente, quantos elementos são promovidos ao todo durante a remoção dos valores 10 e 80.

a. 1
b. 2
c. 0
d. 4
e. 3

Questão **22**

Resposta salva

Vale 1,00 ponto(s).

Marcar questão

Assinale as alternativas que apresentam sequências de valores que, quando inseridos em árvores binárias de busca vazias, resultam em árvores nas quais todos os nós possuem medida de balancemento -1, 0, ou 1.

Nesta questão pode haver mais de uma alternativa correta.

- [ ] a. 4, 1, 3, 2, 5, 6, 7
- [ ] b. 4, 5, 1, 6, 2, 7, 3
- [ ] c. 4, 1, 6, 2, 5, 7, 3
- [x] d. 5, 7, 6, 3, 4, 1, 2
- [x] e. 4, 2, 6, 7, 1, 3, 5

Questão **23**

Resposta salva

Vale 1,00 ponto(s).

Marcar questão

Considere uma árvore AVL, inicialmente vazia, na qual são inseridos os seguintes valores, nesta ordem:

8, 1, 9, 2, 3, 7, 5, 4

Assinale a alternative que indica, corretamente, o total de rotações que são realizadas como resultado de todas as inserções:

- ( ) a. nenhuma rotação
- (•) b. três rotações
- ( ) c. duas rotações
- ( ) d. quatro rotações
- ( ) e. apenas uma rotação

Limpar minha escolha

Questão **24**

Resposta salva

Vale 1,00 ponto(s).

Marcar questão

Considerando a árvore AVL gerada pela inserção dos valores listados na questão anterior, assinale a alternativa que indica, corretamente, a altura da árvore.

- (•) a. 3
- ( ) b. 7
- ( ) c. 6
- ( ) d. 4
- ( ) e. 5

Limpar minha escolha

Questão **25**

Resposta salva

Vale 1,00 ponto(s).

Marcar questão

Se os mesmos valores listados na **Questão 21** tivessem sido inseridos em uma árvore binária de busca convencional (isto é, uma estrutura não balanceada), na mesma ordem, qual seria a altura da árvore resultante?

- ( ) a. 6
- ( ) b. 5
- ( ) c. 7
- ( ) d. 3
- (•) e. 4

Questão **26**

Resposta salva

Vale 1,00 ponto(s).

Marcar questão

Considere as seguintes afirmações sobre árvores AVL:

I - Em uma árvore AVL, o balanceamento de qualquer nó (definido como a diferença entre as alturas das subárvores da esquerda e da direita) deve ser, em termos absolutos, no máximo 2.

II - Em uma árvore AVL, na qual o balanceamento de todos os nós é garantido, tem-se que sua altura será sempre logaritmica em relação a quantidade n de elementos armazenados. Consequentemente, a complexidade assintótica das operações de busca, inserção e remoção será O(lgn).

III - As operações de rotação, em uma árvore AVL, tem o propósito de restaurar o balanceamento de um nó a um nível aceitável. De modo informal, pode-se dizer que uma rotação sobre um nó reduz a altura de uma de suas subárvores (aquela que está profunda demais), enquanto aumenta a altura da outra (aquela que está rasa demais), de modo a equalizar a altura de ambas.

IV - Toda inserção de um novo elemento em uma árvore AVL irá, obrigatoriamente, resultar em uma ou mais rotações.

Estão corretas as afirmações:

- (•) a. II e III
- ( ) b. II, III e IV
- ( ) c. I, II, III e IV
- ( ) d. I, II e III
- ( ) e. III e IV
- ( ) f. II e IV

Limpar minha escolha

Questão **27**

Resposta salva

Vale 1,00 ponto(s).

Marcar questão

Considere uma função que recebe como parâmetro um ponteiro para um nó (representando uma árvore ou subárvore binária, mas não necessariamente de busca), e verifica se tal (sub)árvore é ou não uma árvore binária de busca (ABB), devolvendo um valor booleano como retorno.

Três implementações diferentes para tal função são fornecidas abaixo:

```
//
// Assuma a existência das seguintes funções auxiliares que recebem um nó como
// parâmetro, representando (sub)árvores binárias (não necessariamente de busca):
//
// - tamanho: devolve a quantidade de elementos na (sub)árvore definida a partir do nó
// - menor: devolve o valor do menor elemento na (sub)árvore definida a patir do nó
// - maior: devolve o valor do maior elemento na (sub)árvore definida a patir do nó
//

//////////////
// Versão 1 //
//////////////

Boolean verifica_ABB1(No * raiz){
        if(raiz){
                return  (raiz->esq ? (raiz->valor > raiz->esq->valor) : TRUE) &&
                        (raiz->dir ? (raiz->valor < raiz->dir->valor) : TRUE) &&
                        verifica_ABB1(raiz->esq) &&
                        verifica_ABB1(raiz->dir);
        }
        return TRUE; // uma subárvore binária vazia é uma ABB válida!
}

//////////////
// Versão 2 //
//////////////

Boolean verifica_ABB2(No * raiz){
        if(raiz){
                if(raiz->esq) if(!(raiz->valor > maior(raiz->esq))) return FALSE;
                if(raiz->dir) if(!(raiz->valor < menor(raiz->dir))) return FALSE;
                return verifica_ABB2(raiz->esq) && verifica_ABB2(raiz->dir);
        }
        return TRUE;
}

//////////////
// Versão 3 //
//////////////

void verifica_ABB3_rec(No * raiz, int * a, int * index){
        if(raiz){
                verifica_ABB3_rec(raiz->esq, a, index);
                a[*index] = raiz->valor; (*index)++;
                verifica_ABB3_rec(raiz->dir, a, index);
        }
}
Boolean verifica_ABB3(No * raiz){
        int i = 0;
        int n = tamanho(raiz);
        Elemento * a = (Elemento *) malloc(n * sizeof(Elemento));
        verifica_ABB3_rec(raiz, a, &i);
        for(i = 0; i < n - 1; i++) if(a[i] > a[i + 1]) return FALSE;
        return TRUE;
}
```

Com base nas implementações apresentadas, podemos afirmar que:

- (•) a. as versões 2 e 3 estão corretas.
- ( ) b. Todas as versões estão corretas.
- ( ) c. as versões 1 e 3 estão corretas.
- ( ) d. apenas a versão 3 está correta.
- ( ) e. as versões 1 e 2 estão corretas.
- ( ) f. apenas a versão 2 está correta.
- ( ) g. apenas a versão 1 está correta.

Limpar minha escolha

Questão **28**

Resposta salva

Vale 1,00 ponto(s).

Marcar questão

Suponha que você tenha um vetor (array) com **n** valores inteiros, **sem qualquer tipo de organização prévia**, e que você precise verificar se **3 valores de interesse** (x, y e z) pertencem a este vetor. Assinale a alternativa que descreve a melhor forma (mais eficiente em termos de complexidade assintótica no que diz respeito ao tempo de execução e, no caso de empate, a que dê menos trabalho para implementar) de abordar este problema:

a. Não existe uma abordagem que é claramente superior às demais.

b. Realizar 3 buscas sequenciais diretamente no vetor para verificar a existência (ou não) dos valores x, y e z.

c. Inserir todos os valores do vetor em uma árvore binária de busca, e em seguida realizar 3 buscas na árvore para verificar a existência (ou não) dos valores x, y e z.

d. Ordenar o vetor usando um algoritmo de ordenação que tenha complexidade O(n*log2(n)) e em seguida realizar 3 buscas binárias para verificar a existência (ou não) dos valores x, y e z.

Limpar minha escolha

Questão **29**

Resposta salva

Vale 1,00 ponto(s).

Marcar questão

Suponha que você tenha um vetor (array) com **n** valores inteiros, **sem qualquer tipo de organização prévia**, e que você precise verificar se **n/2 valores de interesse** (x1, x2, x3, ...) pertencem a este vetor. Assinale a alternativa que descreve a melhor forma (mais eficiente em termos de complexidade assintótica no que diz respeito ao tempo de execução e, no caso de empate, a que dê menos trabalho para implementar) de abordar este problema:

a. Ordenar o vetor usando um algoritmo de ordenação que tenha complexidade O(n*log2(n)) e em seguida realizar n/2 buscas binárias para verificar a existência (ou não) dos valores x1, x2, x3, ...

b. Não existe uma abordagem que é claramente superior às demais.

c. Realizar n/2 buscas sequenciais diretamente no vetor para verificar a existência (ou não) dos valores x1, x2, x3, ...

d. Inserir todos os valores do vetor em uma árvore binária de busca, e em seguida realizar n/2 buscas na árvore para verificar a existência (ou não) dos valores x1, x2, x3, ...

Limpar minha escolha

Questão **30**

Resposta salva

Vale 9,00 ponto(s).

Marcar questão

Seu colega, depois de estudar diversos algoritmos de ordenação baseados em comparação, ficou intrigado com o fato de que alguns algoritmos quadráticos (como o insertion sort, e o bubble sort) podem apresentar desempenho linear para algumas sequências de entrada específicas, o que os torna, nestes casos, melhores do que os algoritmos considerados ótimos (como o merge sort, o heap sort e o quicksort).

Com o objetivo de criar um algoritmo que tenha complexidade de tempo **Theta(nlgn)** em casos gerais e complexidade **Theta(n)** para entradas favoráveis (como, por exemplo, sequências que supostamente já deveriam estar ordenadas, mas por algum motivo apresentam um ou outro elemento fora do lugar), seu colega escreveu o seguinte algoritmo:

```
void smart_sort(int * a, int n){

        // a: array contendo os valores a serem ordenados
        // n: tamanho do array / quantidade de elementos a serem ordenados

        int i, trocas, varreduras, max_varreduras;

        varreduras = 0;
        max_varreduras = log2(n);

        do{
                for(trocas = 0, i = 0; i < n - 1; i++){

                        if(a[i] > a[i + 1]) {

                                troca(a, i, i + 1);
                                trocas++;
                        }
                }
                varreduras++;

        } while(trocas > 0 && varreduras < max_varreduras);

        if(trocas > 0) merge_sort(a, 0, n - 1); // versão padrão do merge sort.
}
```

Disserte a respeito do algoritmo smart sort apresentado pelo seu colega. Avalie o comportamento do algoritmo sob o ponto de vista teórico e formal (complexidade assintótica em relação ao tempo de execução), mas também faça considerações do ponto de vista prático. Ao final, conclua dando sua visão sobre o algoritmo: ele cumpre seu objetivo?

O vetor estando ordenado a complexidade assintótica será Θ(n) (Theta(n)), visto que, quando não existe nenhuma troca, apenas uma varredura será feita com n - 1 comparações. Então, no melhor caso que o vetor é ordenado, o smart_sort é Ω(n) (Omega(n)), assim como insertionSort e bubbleSort.

Para o caso de um vetor com apenas um elemento fora de ordem, o algoritmo irá executar Θ(nlgn) (Theta(nlgn)), visto que, o mesmo executará o 'for' n - 1 vezes e o 'while' lg n vezes, e ainda chamará o algoritmo merge_sort. O que é assintoticamente equivalente ao algoritmo merge_sort.

Considerando um vetor totalmente desordenado a complexidade assintótica do algoritmo smart_sort ainda continua sendo Θ(nlgn) (Theta(nlgn)), tanto por, executar 'for' n - 1 vezes e o 'while' lg n vezes, quanto por chamar o algoritmo merge_sort.

Portanto, considerando que o objetivo era ser Ω(n) para casos favoráveis (vetor quase ordenado e ordenado), o algoritmo smart_sort o cumpre parcialmente. Visto que, é Ω(n) (melhor caso) quando o vetor já está ordenado (objetivo cumprido). Já para casos onde o vetor está quase ordenados (objetivo não cumprido) ou completamente desordenado não traz vantagem nenhuma, porque executará Θ(nlgn) (antes de chamar o merge_sort), e chamará a função merge_sort que Θ(nlgn). No ponto de vista prático, devido sua constante, no caso geral o smart_sort é inferior ao merge_sort.