Projeto e Análise de Algoritmos

Lista de exercícios para a A1.

Pedro Santos Tokar - Matrícula: 231708008

Questão 1

Para provar que $T(n)$ pertence ao conjunto $\theta(f(n))$ (escrito comumente como $T(n) = \theta(f(n))$), é necessário provar que $f(n)$ pertence tanto ao conjunto $O(f(n))$ quanto ao conjunto $\Omega(f(n))$. Provar que $T(n) = O(f(n))$ é, por definição, provar que $\exists\ c, n_0$ constantes positivas tais que $T(n) \leq c \cdot f(n) \forall n \geq n_0$. A definição é semelhante para $\Omega(f(n))$, mas inverte-se o sentido da desigualdade.

Item a)

Inspecionando a função, é intuítivo pensar em $f(n) = n^3$. Primeiro, vamos analisar se $T(n) = O(n^3)$, reescrevendo a desigualdade:

$$T(n) \leq c \cdot f(n) \implies 4n^3 + n^2 + 3n \leq c \cdot n^3 \implies 4 + \frac{1}{n} + \frac{3}{n^2} \leq c$$

A medida em que n se torna maior, tanto $\frac{1}{n}$ quanto $\frac{3}{n^2}$ irão diminuir. Tendo conhecimento desse comportamento, podemos definir $c = 5$ e $n_0 = 4$, valores que satisfazem a definição. Portanto, $T(n) = O(n^3)$.

A análise é semelhante para $\Omega(n^3)$: queremos c e n_0 tal que $4 + \frac{1}{n} + \frac{3}{n^2} \geq c$. É fácil ver que podemos usar novamente $n_0 = 4$ e tomar $c = 3$ para tornar a desigualdade verdadeira. Com isso, provamos que $T(n) = \theta(n^3)$.

Item b)

Novamente, por inspeção direta da função, $f(n) = n \log n$ é uma hipotese aceitável. Dessa vez, vamos fazer a análise para $\Omega(n \log n)$ e $O(n \log n)$ ao mesmo tempo:

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n) \implies c_1 \cdot n \log n \leq 2n \log n + 2n + 7 \leq c_2 \cdot n \log n \implies$$
$$\implies c_1 \leq 2 + \frac{2}{\log n} + \frac{7}{n \log n} \leq c_2$$

Se tomamos $n_0 = 2$, o valor da expressão $2 + \frac{2}{\log n} + \frac{7}{n \log n}$ é $\frac{13}{2}$, e para qualquer $n \geq n_0$ esse valor irá diminuir, já que tanto $\frac{2}{\log n}$ quanto $\frac{7}{n \log n}$ diminuem a medida em que n aumenta. Logo, se tomarmos $c_2 = 7$, a segunda desigualdade é verdadeira. Tomar $c_1 = 1$ também satisfaz a primeira, provando assim que $T(n) = \theta(n \log n)$.

Item c)

Antes de encontrar uma função $f(n)$ para testar se ela se adequa, é conveniente reescrever $T(n)$ usando as propriedades dos logarítmos:

$$T(n) = 100 \log(n^5) + n^2 = 100 \cdot 5 \log n + n^2 = n^2 + 500 \log n$$

A escolha evidente de $f(n)$ é, nesse caso, n^2. Testando:

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n) \implies c_1 n^2 \leq n^2 + 500 \log n \leq c_2 n^2 \implies c_1 \leq 1 + 500 \frac{\log n}{n^2} \leq c_2$$

Para analisar o comportamento da expressão $\frac{\log n}{n^2}$ quando $n \to \infty$, podemos usar a regra de L'Hôpital, já que temos uma indeterminação do tipo $\frac{\infty}{\infty}$:

$$\lim_{n \to \infty} \frac{\log n}{n^2} = \lim_{n \to \infty} \frac{\frac{1}{n}}{2n} = \lim_{n \to \infty} \frac{1}{2n^2} = 0$$

Logo, a medida em que n for aumentando, o valor da expressão irá diminuir. Tendo esse conhecimento, podemos tomar $n_0 = 50$, $c_1 = 0.5$ e $c_2 = 5$, e a desigualdade será verdadeira. Assim, sabemos que $T(n) = \theta(n^2)$.

Questão 2

Suponhamos que existe uma função $T(n)$ que seja $O(n^a)$. Nesse caso, sabemos que $\exists\ c_1, n_0$ positivos tais que $T(n) \leq c_1 n^a, \forall\ n \geq n_0$. Agora, vamos supor que essa função também é $\Omega(n^b)$. Pela suposição, para outras constantes c_2, n_0' positivas, teremos que $c_2 n^b \leq T(n), \forall\ n \geq n_0'$. Tomando $n_0'' = \max(n_0, n_0')$, podemos juntar as duas desigualdades em uma única:

$$c_2 n^b \leq T(n) \leq c_1 n^a, \forall n \geq n_0'' \implies c_2 n^b \leq c_1 n^a \implies \frac{n^b}{n^a} \leq \frac{c_1}{c_2} \implies n^{b-a} \leq \frac{c_1}{c_2}, \forall n \geq n_0''$$

Como temos que $b > a$, sabemos que n^{b-a} cresce a medida em que n cresce. Logo, se tomamos $n > \sqrt[b-a]{\frac{c_1}{c_2}}$, a desigualdade será falsa, mostrando que é impossível haver uma função $T(n)$ que é tanto $O(n^a)$ quando $\Omega(n^b)$ com $b > a$.

Questão 3

Pela definição, sabemos que para $\log n$ ser $O(n^a)$, é necessário haver constantes positivas c, n_0 tais que $\log n \leq c \cdot n^a, \forall\ n \geq n_0$. Isso é o mesmo que escrever $\frac{\log n}{n^a} \leq c$. Para analisar o comportamento da expressão $\frac{\log n}{n^a}$ quando $n \to \infty$, podemos usar a regra de L'Hôpital, já que temos uma indeterminação do tipo $\frac{\infty}{\infty}$:

$$\lim_{n\to\infty} \frac{\log n}{n^a} = \lim_{n\to\infty} \frac{\frac{1}{n}}{an^{a-1}} = \lim_{n\to\infty} \frac{1}{an^a} = 0$$

O último passo se apoia no fato de que $a > 0$, o que faz n^a crescer a medida que n cresce. Conhecendo o comportamento decrescente de $\frac{\log n}{n^a}$, podemos afirmar que existem c e n_0 tais que a desigualdade é verdadeira. Mais especificamente, podemos escolher $c = \frac{1}{2^a}$ e $n_0 = 4$ (já que $\frac{\log 4}{4^a} = \frac{1}{2^a}$ <= c) para sempre satisfazer a desigualdade.

Questão 4

Para facilitar a construção da função $T(n)$ referente ao número aproximado de operações do algorítmo, inseri comentários contando as operações realizadas em cada linha:

```
int f(int *array, int lenght) {
    int count = 0;                          //1 operação
    for (int i = 0; i < lenght; i++) {      //fará n vezes o que estiver dentro do
laço
        count += i;                         //1 operação
    }
    // O valor de count será o somatório dos inteiros de 1 até n-1, que é
n*(n-1)/2
    int result = 0;                         //1 operação
    for (int i = 0; i < count; i++) {       //fará n*(n-1)/2 vezes o que estiver
dentro do laço
```

```
        for (int j = 0; j*j < lenght; j++) {//fará sqrt(n) vezes o que estiver
dentro do laço (está dentro do loop de i mas não depende do i)
            result += array[j*j];           //1 operação
        }
    }
    return result;
}
```

Agora, é possível aproximar o número de operações em função de n:

$$T(n) \approx 1 + n + 1 + \frac{n(n-1)}{2}\sqrt{n} = 2 + n + \frac{n^2 - n}{2}\sqrt{n} = \frac{n^{\frac{5}{2}}}{2} - \frac{n^{\frac{3}{2}}}{2} + n + 2$$

Essa função aparenta ser $O\left(n^{\frac{5}{2}}\right)$. Conferindo:

$$\frac{n^{\frac{5}{2}}}{2} - \frac{n^{\frac{3}{2}}}{2} + n + 2 \leq c \cdot n^{\frac{5}{2}} \implies \frac{1}{2} - \frac{1}{2n} + \frac{1}{n^{\frac{3}{2}}} + \frac{2}{n^{\frac{5}{2}}} \leq c$$

É fácil observar que todos os termos, exceto $\frac{1}{2}$, diminuem a medida em que n aumenta. Logo, se tomarmos $n_0 = 4$ e $c = 1$, teremos que a desigualdade é verdadeira $\forall n \geq n_0$, e por consequência $T(n) = O\left(n^{\frac{5}{2}}\right)$, que é a complexidade do algoritmo.

Questão 5

Item a)

A recorrência

$$T(n) = \begin{cases} \theta(1) & \text{se } n = 1 \\ T(n-1) + 2n^2 & \text{se } n > 1 \end{cases}$$

pode ser facilmente resolvida usando o método da iteração, já que não há um multiplicador diferente de 1 multiplicando a função $T(n-1)$:

$$\begin{aligned} T(n) &= T(n-1) + 2n^2 = (T((n-1)-1) + 2(n-1)^2) + 2n^2 = \\ &= T(n-2) + 2(n-1)^2 + 2n^2 = (T((n-2)-1) + 2(n-2)^2) + 2(n-1)^2 + 2n^2 = \\ &= T(n-3) + 2(n-2)^2 + 2(n-1)^2 + 2n^2 = \ldots = \\ &= T(n-k) + 2\sum_{i=0}^{k-1}(n-i)^2 \end{aligned}$$

Como queremos uma fórmula que não tenha recorrências, precisamos encontrar k tal que $n - k = 1$, igualdade satisfeita por $k = n - 1$:

$$T(n) = T(1) + 2\sum_{i=0}^{n-2}(n-i)^2$$

Como $T(1) = \theta(1)$, precisamos nos preocupar apenas com o somatório. Resolvendo-o:

$$2\sum_{i=0}^{n-2}(n-i)^2 = 2\sum_{i=2}^{n} i^2 = 2\frac{n(n+1)(2n+1)}{6} = \frac{2n^3}{3} + n + \frac{1}{3}$$

Como mostrado no primeiro exercício com uma função polinomial semelhante, essa expressão é $O(n^3)$.

Item b)

A recorrência é da forma

$$T(n) = \begin{cases} \theta(1) & \text{se } n = 1 \\ aT\left(\frac{n}{b}\right) + O(n^c) & \text{se } n > 1 \end{cases}$$

Esse tipo de recorrência é favoravel para aplicar o método mestre, tendo em vista que comparar duas funções polinomiais não requer análises mais complexas: para saber qual é polinomialmente maior do que a outra, basta comparar saeus expoentes. Como $\log_3 2$ é menor que 1, temos que $3 > \log_3 2$ e consequentemente $n^3 > n^{\log_3 2}$, ou seja, $n^3 = \Omega\left(n^{\log_3 2}\right)$.

Como esse é o terceiro caso do teorema mestre, é necessário verificar se existem $c < 1$ e n_0 tais que $2\left(\frac{n}{3}\right)^3 \leq cn^3$:

$$2\left(\frac{n}{3}\right)^3 \leq cn^3 \Longrightarrow \frac{2}{9} \leq c$$

Como eles existem, podemos afirmar que $T(n) = \theta(n^3)$.

Item c)

Essa recorrência é semelhante a encontrada no exercício anterior, o que indica que é possível aplicar o método mestre, com $f(n) = 9\sqrt{n} = 9n^{\frac{1}{2}}$. Como $n^{\log_4 4} = n^1$ e n é polinomialmente maior do que $\sqrt{n}$, temos que $\sqrt{n}$ é $O(n)$, então a recorrência cai no primeiro caso do método mestre, e por consequência é $O(n)$.

Vamos conferir se um resultado similar é encontrado aplicando o método da árvore de recursão e o método da iteração. Começando pelo método da iteração:

$$\begin{aligned} T(n) &= 4T\left(\frac{n}{4}\right) + 9\sqrt{n} = 4\left(4T\left(\frac{n}{16}\right) + 9\sqrt{\frac{n}{4}}\right) + 9\sqrt{n} = \\ &= 16T\left(\frac{n}{16}\right) + 4 \cdot \frac{9}{2}\sqrt{n} + 9\sqrt{n} = 16\left(4T\left(\frac{n}{64}\right) + 9\sqrt{\frac{n}{16}}\right) + 2 \cdot 9\sqrt{n} + 9\sqrt{n} = \\ &= 64T\left(\frac{n}{64}\right) + 16 \cdot \frac{9}{4}\sqrt{n} + 2 \cdot 9\sqrt{n} + 9\sqrt{n} = \ldots = \\ &= 4^k T\left(\frac{n}{4^k}\right) + 9\sqrt{n}\sum_{i=0}^{k-1} 2^i \end{aligned}$$

Como queremos uma fórmula que não tenha recorrências, precisamos encontrar k tal que $\frac{n}{4^k} = 1$, igualdade satisfeita por $k = \log_4 n$. Substituindo k por $\log_4 n$, teremos:

$$T(n) = 4^{\log_4 n} T\left(\frac{n}{4^{\log_4 n}}\right) + 9\sqrt{n}\sum_{i=0}^{\log_4 n - 1} 2^i = n + 9\sqrt{n}\sum_{i=0}^{\log_4 n - 1} 2^i$$

Para solucionar o somatório, é importante se atentar à igualdade $2^{\log_4 n} = 2^{\frac{\log_2 n}{2} - 1} = \frac{\sqrt{n}}{2}$. Sabendo dela, podemos expandir o somatório:

$$\sum_{i=0}^{\log_4 n} 2^i = 1 + 2 + 4 + ... + \frac{\sqrt{n}}{2} = \sqrt{n} - 1$$

Fazendo a substituição na fórmula para $T(n)$, teremos que $T(n) = n + 9n - 9\sqrt{n}$. É evidente, então, que $T(n) = O(n)$.

Os resultados obtidos pelo método da árvore de recursão são bem parecidos. No nível 0, a árvore terá apenas 1 nó, que processará n elementos com custo $9\sqrt{n}$. No nível 1, ela terá 4 nós, que processarão $\frac{n}{4}$ elementos cada, com custo por nó de $9\sqrt{\frac{n}{4}}$. No nível 2, ela terá 16 nós, que processarão $\frac{n}{16}$ elementos cada, com custo por nó de $9\sqrt{\frac{n}{16}}$. Seguindo esse padrão, o nível k terá 4^k nós, com custo por nó de $9\sqrt{\frac{n}{4^k}}$.

A altura que a árvore deverá alcançar deve satisfazer $4^k = n$, ou seja, $k = \log_4 n$. Com essas informações, podemos montar a fórmula $T(n)$ somando os custos de todos os níveis:

$$T(n) = n + \sum_{i=0}^{\log_4 n - 1} 9 \cdot 4^i \sqrt{\frac{n}{4^i}} = n + 9 \sum_{i=0}^{\log_4 n - 1} \frac{4^i}{2^i}\sqrt{n} = n + 9\sqrt{n} \sum_{i=0}^{\log_4 n - 1} 2^i = 10n - 9\sqrt{n}$$

A fórmula encontrada foi a mesma do método da recursão, e por isso a complexidade dada por esse método é compatível com a dos demais: $O(n)$.

A melhor forma de verificar esse resultado, porém, é usando o método da substituição, que faz a prova por indução usando a definição. Primeiro, vamos analisar o caso base $n = 1$:

$$T(n) = 1 \leq 1c, \text{verdadeiro para } c \geq 1$$

Agora, analisando o caso $n = \frac{n}{4}$:

$$\begin{aligned} T\Big(\frac{n}{4}\Big) \leq c\Big(\frac{n}{4}\Big) \implies \frac{T(n) - 9\sqrt{n}}{4} \leq c\frac{n}{4} \leq cn \implies \\ \implies T(n) - 9\sqrt{n} \leq cn \leq 4cn \implies \\ \implies cn + 9\sqrt{n} \leq 4cn \implies \\ \implies 9\sqrt{n} \leq 3cn, \text{satisfeito se } c \geq 3 \text{ e } n_0 = 9. \end{aligned}$$

Portanto, T(n) é O(n).

Item d)

Para essa recorrência, podemos aplicar o método mestre. Precisamos observar que $\log_5 6 > 1$, o que nos leva a concluir que $f(n) = n \log n = O\big(n^{\log_5 6 - \varepsilon}\big)$ para $\varepsilon = 0,05$ (pois $\log_5 6 = 1,113...$). Esse é o primeiro caso do teorema mestre, o que leva a conclusão de que $T(n) = \theta\big(n^{\log_5 6}\big)$.

Item e)

Podemos aplicar o método da árvore de recursão para resolver essa recorrência. O nó pai, no nível 0, terá custo de cn. No nível 1, teremos três nós: dois com $\frac{n}{4}$ elementos e um com $\frac{n}{2}$ elementos. É importante observar que o custo desse nível e dos níveis seguintes, quando somado, será igual a n, já que não se perdem elementos de nível para nível. Como a árvore é desbalanceada e estamos tentando encontrar um limite superior, é possível usar a altura máxima da árvore para calcular o custo total dela.

A altura máxima da árvore será atingida pelos nós que carregam $\frac{n}{2}$ elementos de seus pais. Isso indica que a árvore terá altura máxima de $\log_2 n$. Como o custo de cada nível é n, teremos então que o limite superior é $O(n \log n)$.

Questão 6

Para facilitar a construção da função $T(n)$ referente ao número aproximado de operações do algorítmo, inseri comentários contando as operações realizadas em cada linha:

```
int f(int *array, int start, int end) {
    int size = end - start; //Calcula o n para essa chamada da função
    if (size <= 0) { //Se n = 1, trata do caso base
        return 0; //Em theta(1)
    }

    int mid = (start + end) / 2; //Divide n em 2
    int leftCount = f(array, start, mid); //Recursivamente chama pra metade
esquerda
    int rightCount = f(array, mid + 1, end); //E para a metade direita
    int count = leftCount + rightCount; //Operação em theta(1)

    for (int i = 0; pow(i, 2) < size; i++) { //Itera sqrt(n) vezes
        for (int j = 0; pow(2, j) < size; j++) { //Itera logn vezes
            count++; //Operação em theta(1)
        }
    }
    return count;
}
```

Agora, é possível reconstruir a recorrência que ditará a complexidade dessa função:

$$T(n) = \begin{cases} \theta(1) & \text{se } n = 1 \\ 2T(\frac{n}{2}) + \sqrt{n}\log_2 n & \text{se } n > 1 \end{cases}$$

É possível resolver essa recorrência usando o método mestre. Observamos que $\log_2 2 = 1 \Rightarrow n^{\log_2 2} = n$, e que $\sqrt{n}\log n = O(n^{1-0.1})$, caindo no primeiro caso do teorema mestre. O resultado obtido, então, é que $T(n) = \theta(n)$.

Questões 7 a 15 - Observações

Todas as questões a seguir foram implementadas em C++. Como a linguagem carece de uma hashtable por padrão, implementei uma com uma implementação bem semelhante à passada pelo professor em aula. O tipo de Hashtable implementado foi a que resolve conflitos por listas encadeadas, já que evitaria a necessidade de implementar a sondagem linear/quadrática. A implementação é a apresentada a seguir:

```
typedef struct HashTableNode HTNode;
struct HashTableNode {
    unsigned key;
    int value;
    HTNode* next;
    HTNode* previous;
};

class HashTable {
```

```
private:
    HTNode** m_table; //An array with the respective lists for each hash
    int m_size; //The size of the hash function image set;

    unsigned hash(unsigned key) {
        return key % m_size;
    }

public:
    HashTable(int iM): m_table(nullptr), m_size(iM) { //iM is hash image size
        m_table = new HTNode*[iM]; //initializes as an array (before initialized
as nullptr)
        for (int i = 0; i < m_size; i++) { //will set all the values to nullptr
since no insertion has been made
            m_table[i] = nullptr;
        }
    }

    ~HashTable() {
        for (int i = 0; i < m_size; i++) { //Will delete every hash stored values
            HTNode * node = m_table[i];
            while (node != nullptr) { //Delete the entire linked list of that hash
value
                HTNode* nextNode = node->next;
                delete node;
                node = nextNode;
            }
        }
        delete[] m_table; //Delete hashes array
    }

    void set(unsigned key, int value){ //For seting a pair of key, value
        unsigned hashed = hash(key); //Gets the hash for the key
        HTNode* ptrNode = m_table[hashed]; //Gets the linked list of that hash
        while (ptrNode != nullptr and ptrNode->key != key){ //And runs till find
the node repecting to the key
            ptrNode = ptrNode->next;
        }
        if (ptrNode == nullptr){ //The key has never been used, so creates the
node
            ptrNode = new HTNode;
            ptrNode->key = key;
            ptrNode->previous = nullptr;
            ptrNode->next = m_table[hashed];
            if (m_table[hashed] != nullptr){
                m_table[hashed]->previous = ptrNode;
            } //Inserts before the list (doesn't need to add in the end)
            m_table[hashed] = ptrNode;
        }
        ptrNode->value = value; //in either cases, update the value
    }

    HTNode* get(unsigned key){
        unsigned hashed = hash(key);
        HTNode* ptrNode = m_table[hashed];
```

```
        while (ptrNode != nullptr and ptrNode->key != key){ //And runs till find
the node repecting to the key
            ptrNode = ptrNode->next;
        }
        return ptrNode;
    }

    void remove(unsigned key){
        unsigned hashed = hash(key);
        HTNode* ptrNode = m_table[hashed];
        while (ptrNode != nullptr and ptrNode->key != key){ //And runs till find
the node repecting to the key
            ptrNode = ptrNode->next;
        }
        if (ptrNode != nullptr){
            HTNode* nextNode = ptrNode->next;
            if (nextNode != nullptr) {
                nextNode->previous = ptrNode->previous;
            }
            HTNode* previousNode = ptrNode->previous;
            if (previousNode != nullptr) {
                ptrNode->previous->next = ptrNode->next;
            } else {
                m_table[hashed] = ptrNode->next;
            }
            delete ptrNode;
        } //if reaches here, then the key did'nt exist anyway
    }
};
```

Assim como orientado pelo monitor, as operações de consulta, inserção, atualização e remoção de chaves e valores estão sendo contabilizadas como tendo complexidade $O(1)$, mesmo que na prática a função de hash implementada seja bem simples.

A função de distância entre dois números ($|x_1 - x_2|$) também é usada em diversas questões, e foi implementada com complexidade $\theta(1)$:

```
template <typename T>
T dist(T x1, T x2){
    T result = x1 - x2;
    if (result < 0){
        result = result * -1;
    }
    return result;
}
```

Também foram usadas árvores binárias em algumas questões. A implementação de uma árvore binária é bem simplória:

```
template <typename T>
struct NodeTree
{
    T iPayload;
    NodeTree* ptrLeft;
```

```
    NodeTree* ptrRight;
};

template <typename T>
NodeTree<T>* newNodeTree(T iData)
{
    NodeTree<T>* tmp = (NodeTree<T>*) malloc(sizeof(NodeTree<T>));

    if (tmp != nullptr)
    {
        tmp->iPayload = iData;
        tmp->ptrLeft = nullptr;
        tmp->ptrRight = nullptr;
    }

    return tmp;
}
```

A criação de um nó é feita em tempo constante $\theta(1)$

Um último detalhe importante a se adicionar é que a maioria dos algorítmos faz uso da Mediana das Medianas para encontrar a mediana de sequências e particioná-las a partir disso. Teoricamente, isso sempre parte a array de elementos em duas partes iguais. Porem, na implementação, isso não acontece se o elemento correspondente à mediana estiver duplicado e aparecer antes do meio.

Esse é um problema solucionado com algumas condicionais, que acabam deixando os algorítmos em si com menos destaque. Como o intuíto é mostrar os algorítmos, essa correção foi omitida dos trechos de programas em C++.

Questão 7

Para implementar o algorítmo, se fez necessário uma versão modificada do algorítmo Mediana das Medianas, que fosse capaz de fazer as comparações baseadas no valor retornado pela hashtable com um inteiro como chave (por consequência, o algorítmo de partição também foi modificado para funcionar com hashtables). Como acessar um valor na hashtable é constante, trocar uma comparação $A[i] < A[j]$ por $H[A[i]] < H[A[j]]$ não mudou a complexidade do algorítmo Mediana das Medianas, que seguiu sendo $O(n)$.

O algorítmo que resolve o problema da questão em si é o apresentado abaixo:

```
int ex7(int vector[], int iLength, int k){
    int integers[iLength]; //array que armazenará todos os inteiros únicos que
aparecem na sequência A
    int m = 0;
    HashTable* freqs = new HashTable(iLength); //Hashtable que mapeará inteiro ->
frequência de aparição
    for (int i = 0; i <iLength; i++){ //Repetindo operações n vezes
        if(freqs->get(vector[i]) == nullptr){ //Se não tinha passado por aquele
inteiro, atualiza m e inicia a frequência como 1
            integers[m] = vector[i];
            m++;
            freqs->set(vector[i], 1);
        } else { //Se já tinha passado, apenas atualiza a frequência
```

```
            freqs->set(vector[i], freqs->get(vector[i])->value + 1);
        }
    } //As operações feitas foram todas O(1), então a complexidade desse loop é
O(n)
    int x = MOMSelect(integers, 0, m, m - k, freqs); //Como visto, MOM faz seleção
em tempo linear O(n). É buscado m - k já que queremos o k-ésimo *mais* frequente
    return x;
}
```

Ele é composto de duas principais operações, sendo a primeira a contagem das frequências das aparições que os inteiros fazem na lista A.

Essa contagem é feita de forma semelhante ao que acontece no algorítmo Counting Sort, mas no lugar de usarmos uma array, usamos uma hashtable, já que não necessariamente teremos todos os inteiros menores que m. Como acessar os valores da hashtable é uma operação com tempo constante, construir as frequências é feito em tempo linear $O(n)$.

Com as frequências em mãos e uma array com os inteiros que aparecem (cada um aparecendo uma única vez), a segunda etapa consiste em encontrar o k-ésimo inteiro que mais aparece. Como explicado acima, isso é feito usando uma versão modificada do Mediana das Medianas, que terá complexidade $O(m)$. Como $m \leq n$, $O(m) \leq O(n)$

Logo, o custo total do algorítmo é $T(n) = O(n) + O(m) + \theta(1) = O(n)$.

Questão 8

O algorítmo que soluciona o exercício é apresentado abaixo:

```
bool ex8(int vector[], int iLength, int x, int result[]){
    HashTable* appears = new HashTable(iLength); //essa hashtable irá contar se um
número aparece ou não na lista A
    for (int i = 0; i < iLength; i++){ //Repetindo operações n vezes
        if (appears->get(vector[i]) == nullptr){ //Se o número não havia
aparecido, marca a aparição dele
            appears->set(vector[i], 1);
        }
    } //Ao final desse for, a hashtable dirá se um número apareceu ou não
(semelhante a um set de python)
    for (int i = 0; i < iLength; i++){ //Repetindo operações n vezes
        if (appears->get(x - vector[i]) != nullptr){ //Para cada número, verifica
se o valor x - número existe na lista
            result[0] = vector[i];
            result[1] = x - vector[i];
            return true; //Se existir, marca que existe
        }
    } //Se chegou até aqui, significa que para nenhum número existe outro na array
que some x
    return false;
}
```

Novamente, o algorítmo é dividido em duas etapas principais. A primeira é montar uma hashtable que diz se um número aparece ou não na lista A. A construção dessa hashtable é $O(n)$, já que a lista é percorrida uma vez.

A segunda etapa consiste em verificar, para todos os números, se existe outro na array de tal forma que os dois somem dois. Para fazer essa verificação, é usado o fato de que $x = x + n - n = n + (x - n)$ para qualquer n. Logo, se tanto n quanto $x - n$ aparecerem na lista, sabemos que há uma solução, e retornamos ela. Essa etapa percorre a lista uma vez no pior caso, então sua complexidade é $O(n)$. Por consequência, o custo total do algorítmo é $T(n) = O(n) + O(n) + \theta(1) = O(n)$

Questão 9

Por implementar o algorítmo em C++, alguns contornos tiveram que ser feitos para poder tratar uma lista como matriz, já que não é possível passar uma array de arrays para uma função nessa linguagem sem mexer com ponteiros. Por isso, a função recebe uma lista de tamanho $n \cdot m$, m e n. A indexação é feita para simular acessar uma array dentro de outra. Na prática, não prejudica o algorítmo.

Ele é o apresentado abaixo:

```
int ex9(int vector[], int iCount, int iLength){
    int iDisjuncts = 0;
    HashTable* freqs = new HashTable(iLength * iCount); //hashtable que terá as
frequências
    for (int i = 0; i < iCount; i++){
        for (int j = 0; j < iLength; j++){
            if(freqs->get(vector[i*iCount + i + j]) == nullptr){ //Inicializa a
chave se necessário
                freqs->set(vector[i*iCount + i + j], 1);
            } else { //Adiciona pras frequências
                freqs->set(vector[i*iCount + i + j], freqs->get(vector[i*iCount +
i + j])->value + 1);
            }
        }
    }
    for (int i = 0; i < iCount; i++){ //Vai realizar essas operações para todas as
listas
        bool bIntersects = false;
        HashTable* aux = new HashTable(iLength);
        for (int j = 0; j < iLength; j++){ //Novamente constrói a hashtable de
frequências, só para essa lista
            if(aux->get(vector[i*iCount + i + j]) == nullptr){
                aux->set(vector[i*iCount + i + j], 1);
            } else {
                aux->set(vector[i*iCount + i + j], aux->get(vector[i*iCount + i +
j])->value + 1);
            }
        }
        for (int j = 0; j < iLength; j++){ //Agora verifica se os elementos
aparecem em outras listas
            if(aux->get(vector[i*iCount + i + j])->value < freqs-
>get(vector[i*iCount + i + j])->value){
                bIntersects = true;
                break; //Se aparece, já sabe que não é disjunta e não precisa
verificar os demais
            }
        }
```

```
        if(not bIntersects){ //Se é disjunta, aumenta a contagem
            iDisjuncts++;
        }
        delete aux; //Reinicia a hashtable
    }
    return iDisjuncts;
}
```

O algorítmo inicializa, incialmente, a contagem de listas disjuntas e uma hashtable que armazenará a contagem de vezes que um número aparece em qualquer uma das listas. Para efetivamente montar essa hashtable, inserindo os valores, é necessário iterar por todos os elementos de todas as listas, operação que tem custo $O(m \cdot n)$, já que são percorridos n elementos por lista, e são n listas.

A próxima etapa consiste em, por lista:

- Criar uma hashtable de frequências apenas da lista e populá-la (a complexidade será $O(n)$, já que vamos percorrer a lista uma vez);
- Novavmente iterar pelos valores, verificando se globalmente (em todas as listas) o valor aparece mais vezes que na lista atual. Se aparece, significa que o valor se repete em outra lista, e por consequência ela não é disjunta. Novamente, a complexidade é $O(n)$;
- Limpa o dicionário. Como, para limpar, é necessário apagar os elementos, essa operação tem complexidade $O(n)$.

Como podemos concluir, a complexidade do algorítmo será $T(n) = O(n \cdot m) + O(n \cdot m) = O(n \cdot m)$.

Questão 10

O exercício 10 foi implementado como uma variante do algorítmo de busca Mediana das Medianas. Antes de ver o algorítmo, é importante ter em mente que, em uma array particionada, o pivô x irá satisfazer a condição dada (de ter x ou mais elementos maiores ou iguais a ele na array) caso a diferença entre n e o index de x (0-indexado) seja maior que x. Essa verificação será usada durante o algorítmo, já que diversos particionamentos na array são feitos:

```
int ex10(int vector[], int iStart, int iEnd, int iRealEnd){ //iRealEnd é o tamanho
real da array, necessário mesmo dentro de chamadas recursivas. Os outros
parâmetros são sempre fornecidos em valores absolutos na array
    if (iStart == iEnd - 1){ //Se recebe array de apenas um elemento (indice
iStart)
        if ((iRealEnd - iStart) >= vector[iStart]){ //Caso o elemento satisfaça a
condição, retorna ele
            return vector[iStart];
        } else { //Caso contrário, retorna -1
            return -1;
        }
    } //Agora inicia o caso de uma array com mais de um elemento
    int iMedian = MOMSelect(vector, iStart, iEnd, iStart + (iEnd - iStart)/2); //
Primeiro vai encontrar a mediana da array
    int iPivot = partitionate(vector, iStart, iEnd - 1, iMedian); //Depois, vai
particionar ela de forma que a mediana fique no meio dela
    if ((iRealEnd - iPivot) >= iMedian){ //Se a mediana atende o critério,
precisamos procurar por algum possível número que atenda e seja maior. Então,
vamos procurar nos elementos maiores que a mediana atual encontrada
```

```
        int iCompare = ex10(vector, iPivot + 1, iEnd, iRealEnd);
        return max(iCompare, iMedian); //Retornará o máximo (já que podemos ter um
retorno -1 caso não hajam elementos maiores que satisfaçam a condição)
    } else {
        int iCompare = ex10(vector, iStart, iPivot, iRealEnd); //Se o número não
atende, nenhum na frente dele atenderá, logo devemos procurar entre os menores que
ele
        return max(iCompare, -1);
    }
}
```

O algorítmo começa pelo caso base, que verifica se a condição é verdadeira para apenas um elemento da array. Sempre é possível acessar o tamanho original da array, já que essa informação influência na solução. O caso base só faz comparações, então sua complexidade é $\theta(1)$.

Em entradas que representam pedaços maiores da array, o algorítmo:
- Executa a busca da mediana usando Mediana das Medianas, operação com complexidade $O(n)$
- Particiona o pedaço da array, também em $O(n)$
- Executa recursivamente para metade da array (garantido devido ao fato de particionarmos na mediana)
- Após executar, retorna o maior dos valores, garantindo que ao final teremos o maior x possível.

Com essas informações, é possível montar a recorrência do algorítmo:

$$T(n) = \begin{cases} \theta(1) & \text{se } n = 1 \\ O(n) + T(\frac{n}{2}) & \text{se } n > 1 \end{cases}$$

Pelo método da iteração, temos:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + cn = \left(T\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn = \\ &= T\left(\frac{n}{4}\right) + c\frac{n}{2} + cn = \left(T\left(\frac{n}{8}\right) + c\frac{n}{4}\right) + c\frac{n}{2} + cn = \\ &= T\left(\frac{n}{2^k}\right) + cn\sum_{i=0}^{k-1} 2^{-i} \end{aligned}$$

Para termos $\frac{n}{2^k} = 1$, precisamos de $k = \log_2 n$. Fazendo a substituição:

$$T(n) = T(1) + cn\sum_{i=0}^{\log n - 1} 2^{-i} = \theta(1) + cn\left(1 + \frac{1}{2} + \frac{1}{4} + ... + \frac{1}{\frac{n}{2}}\right) < O(n)$$

Assim, encontramos que a complexidade do algorítmo é $O(n)$.

Questão 11

Para entender a solução do exercício, é preciso se atentar para um fato interessante: em uma BST, atravessar por profundidade e em ordem (depth-first-search in order) faz com que os nós sejam visitados em ordem crescente. Fazer uma sondagem desse tipo é uma operação em tempo linear $O(n)$, já que cada nó é visitado apenas uma vez. Então, para realizar o exercício, foi criada uma função auxiliar que cria uma array com os elementos da BST, em ordem crescente, usando a propriedade citada:

```
template <typename T>
int dfsFlatten(T vector[], NodeTree<T>* ptrStartingNode, int iIndex){
    if (ptrStartingNode != nullptr){ //Para evitar problemas de ponteiros nulos
        iIndex = dfsFlatten(vector, ptrStartingNode->ptrLeft, iIndex); //Chama
para o nó da esquerda, e pega o index atualizado
        vector[iIndex] = ptrStartingNode->iPayload; //Adiciona na lista o elemento
do nó atual
        iIndex++; //E atualiza o index
        iIndex = dfsFlatten(vector, ptrStartingNode->ptrRight, iIndex); //Chama
para o nó da esquerda
    }
    return iIndex; //Retorna o index atualizado
}
```

Como dito, a complexidade dessa operação é $O(n)$. Quando temos uma lista ordenada, sabemos que as menores distâncias entre elementos serão alcançadas com elementos adjacentes: elementos não adjacentes terão valores entre eles que por sua vez terão distâncias menores para eles. Por isso, em uma lista ordenada, apenas uma passagem pela lista já é o suficiente para encontrar a menor distância, assim como é feito no algorítmo:

```
int ex11(NodeTree<int>* ptrRoot, int iSize){
    int vector[iSize]; //Array auxiliar que terá todos os valores da BST ordenados
    dfsFlatten(vector, ptrRoot, 0); //Função que insere a BST em uma array
ordenada
    int minDistance = 2147483647; //Maior valor que uma int pode assumir, imagine
um infinito aqui
    for(int i = 1; i < iSize; i++){ //Roda pela lista ordenada
        if (vector[i] - vector[i - 1] <= minDistance){ //Salva a menor diferença
vista até o momento
            minDistance = vector[i] - vector[i - 1];
        }
    }
    return minDistance;
}
```

Como é possível ver, é realizada a operação de colocar a BST em uma array, em tempo linear $O(n)$, e depois a lista é visitada uma vez, procurando quais as menores distâncias entre dois pares de elementos, também em tempo linear $O(n)$. Então, a complexidade final será $T(n) = O(n) + O(n) = O(n)$.

Questão 12

Para fazer esse algorítmo, foi necessário implementar uma versão modificada do algorítmo de Merge Sort e do algorítmo Mediana das Medianas (e consequentemente do algorítmo de particionamento). Mais especificamente, os algorítmos não fazem mais comparações de $A[i] > A[j]$, e sim comparações de $\text{dist}(A[i], x) > \text{dist}(A[j], x)$ (modificações semelhantes às feitas para outras questões, com hashtables). A implementação do exercício é a seguinte:

```
void ex12(int vector[], int iLength, int x, int k, int result[]){
    if (k > iLength) { //Nesse caso, retorna toda a lista ordenada pelo critério
da distância (k = n)
```

```
        k = iLength;
    }
    int iPivot = MOMSelect(vector, 0, iLength, k, x); //Seleciona o késimo mais
próximo de x em tempo linear
    partitionate(vector, 0, iLength - 1, iPivot, x); //Particiona a array,
garantindo que todos que forem mais próximos que o késimo estejam no começo da
array
    mergeSort(vector, 0, k, x); //Ordena apenas os k primeiros elementos da array
    for (int i = 0; i < k; i++){ //Coloca na array de resultados (específico da
linguagem), que vai conter os k primeiros ordenados por proximidade
        result[i] = vector[i];
    }
}
```

O algorítmo começa encontrando qual o k-ésimo número mais próximo do valor de x. Como essa seleção é feita usando Mediana das Medianas, a complexidade de tempo é $O(n)$. A próxima etapa é particionar o vetor no pivô obtido, usando também o critério de distância para x. Como esse algorítmo percorre a lista uma vez, a complexidade dessa etapa também é $O(n)$.

A última etapa realizada é a ordenação dos primeiros k elementos (que já sabemos que são os que devem ser retornados). O algorítmo usado para ordenar é o Merge Sort, já que mesmo em pior caso ele tem complexidade $\theta(n \log n)$. Como ele é executado para os primeiros k elementos, sua complexidade será $\theta(k \log k)$. Somando tudo, teremos que a complexidade do algorítmo será:

$$T(n) = O(n) + O(n) + O(k \log k) \leq O(n) + O(n \log k) = O(n \log k)$$

Questão 13

O algorítmo que resolve o exercício é apresentado abaixo:

```
int ex13(int vector[], int iLength, int x, int result[]){
    heapSort(vector, iLength); //Inicia ordenando a lista
    int bestPair[2] = {0, 0}; //Mantém registro do melhor par i, j
    int i = 0;
    int j = 1; //starts with only sublist [0]
    for(int j = 1; j <= iLength; j++){ //Avança j
        while(vector[j - 1] - vector[i] > x && j > i){ //Avança i até A[i] e A[j]
terem distância menor que x (ou até não dar pra avançar i)
            i++;
        }
        if(j - i >= bestPair[1] - bestPair[0]){ //Avalia se o par é melhor ou pior
            bestPair[1] = j;
            bestPair[0] = i;
        }
    }
    for (int i = bestPair[0]; i < bestPair[1]; i++){
        result[i - bestPair[0]] = vector[i]; //Insere na lista de resultado (c++
problems)
    }
    return (bestPair[1] - bestPair[0] > 1) ? bestPair[1] - bestPair[0] : 0; //Se o
tamanho é 1, significa que não há sublista, então retorna 0 nesse caso
  }
```

Em uma lista ordenada, podemos perceber que se dois elementos tem distância entre eles menor que x, todos os elementos entre eles também terão distâncias menores que x de um pro outro. Com essa propriedade, temos que encontrar os elementos mais afastados um do outro que tenham distância menor que x por consequência encontra a maior subsequência com a propriedade desejada.

Por isso primeiro passo do algorítmo é fazer a ordenação da lista A. Como ela é feita pelo algorítmo Heap Sort, temos que a complexidade mesmo no pior caso de ordenação será $O(n \log n)$.

Para encontrar o par (i, j) com i e j o mais afastados possível e com distância entre os elementos menor que x, o que é feito é:

- Avançar j em uma casa, a cada passo;
- Avançar i até que o elemento tenha distância menor que x do elemento em j. Avançar depois de alcançar o primeiro não daria a maior distância possível, então i para de avançar (se nenhum elemebto dá um adistância desejada, i para antes de alcançar j);
- Avaliar se i e j estão mais distantes que o melhor último par. Se estiverem, definí-los como o melhor par.

É possível perceber, então, que i irá avançar do início até o final da lista, e j, em pior caso, também. Logo, as operações realizadas nessa etapa tem complexidade $O(n)$.

A complexidade do algorítmo então é dada por $T(n) = O(n \log n) + O(n) = O(n \log n)$.

Questão 14 - terminar

O algorítmo guloso necessário para resolver o exercício é apresentado abaixo:

```
void ex14(int vector[], int iLength, int result[]){
    int i = 0;
    int j = iLength - 1; //Inicia i no começo e j no final da lista
    int maxValue = -2147483647; //Maior valor visto até o momento
    while (i < j){
        if ((dist(i, j) * min(vector[i], vector[j]) >= maxValue)) { //Se o novo
valor for melhor, toma nota dele
            maxValue = (dist(i, j) * min(vector[i], vector[j]));
            result[0] = i;
            result[1] = j;
        }
        if (vector[i] <= vector[j]) { //Aqui o índice que estiver no menor
elemento vai avançar na direção da metade da lista
            i++;
        } else {
            j--;
        }
    }
}
```

O funcionamento do algorítmo é explicado nos comentários. Percebe-se que, ao final das iterações, i e j se encontram, e eles só andam para a metade da lista. Logo, serão feitos exatamente n iterações, o que faz a complexidade do algorítmo ser $T(n) = O(n)$.

Para provar que o algorítmo funciona e que faz escolhas ótimas a cada iteração, precisamos observar que:

- A cada etapa, estamos aproximando i de j, ou seja, a cada iteração, o valor $|i - j|$ sempre irá diminuir, independente de demais fatores.
- O critério para determinar se i irá avançar ou j retroceder é qual está associado ao menor elemento. O intuito dessa escolha é explicado a seguir.

Quando iniciamos os ponteiros em extremidades opostas, estamos com i e j no único (e por consequência, melhor) caso para a distância $|i - j| = n$. Temos, então, duas escolhas: avançar i ou retroceder j. Por consequência, teremos apenas dois possíveis resultados finais para o termo $\min\{\cdot\}$: $\min\{A[i], A[j-1]\}$ ou $\min\{A[i+1], A[j]\}$. Queremos aquele que seja maior, ou seja, o par que tiver os maiores elementos. Quando movemos o ponteiro associado ao menor valor, estamos em pior caso obtendo uma dupla com mínimo menor, mas máximo igual. Se mudamos o ponteiro associado ao maior valor, no pior caso obtemos uma dupla com tanto máximo quanto mínimos piores que os originais.

Por isso, **a escolha feita é ótima**: ela encontra um par com mesmo ou maior máximo do que o atual. Após fazer essa escolha, encaramos o mesmo problema novamente, mas para $|i - j| = n - 1$, e já iniciaremos com a opção ótima para essa restrição, e sabendo tomar a melhor escolha novamente, já que a configuração do problema não muda.

Observa-se, então, que em todas as etapas encontramos o melhor par que satisfaz $|i + j| = n - k$ (com k sendo o número de iterações). Como a cada etapa comparamos com o antigo melhor caso, garantiremos que escolheremos, dentre os valores ótimos de cada etapa, o melhor valor. Por consequência, será o melhor valor de toda a lista. Logo, o algorítmo traz a solução ótima, cumprindo com os requisitosa do exercício.

Questão 15

A implementação por dividir e conquistar desse algorítmo é bem intuitiva. Sabemos que, em uma bst, todos os nós de uma subárvore a direita de um nó terão valores maiores que esse nó, e todos os nós de uma subárvore a esquerda terão valores menores que esse nó. Logo, podemos transformar o problema em dividir a lista em valores menores que a mediana e maiores que a mediana, usando particionamento e seleção. Podemos então recursivamente tratar das listas resultantes.

A **etapa de combinação** consiste em montar um nó pai, usando a mediana, e anexar a ele os nós a esquerda e a direita, que são retornados pela própria função recursiva. O **caso base** é quando a sublista tem apenas 1 ou 2 elementos. Para 1 elemento, basta retornar um nó com o valor do elemento. Para dois elementos, é possível construir um nó com um filho, fazendo a seleção certa dos valores. Em ambos casos, as operações são realizadas com tempo constante.

O algorítmo está apresentado abaixo:

```
NodeTree<int>* ex15(int vector[], int iStart, int iEnd){
    int iLength = iEnd - iStart; //Calcula o tamanho da entrada
    if (iLength == 1){ //Trata do caso base de ser apenas 1 número
        return newNodeTree(vector[iStart]); //Nesse caso, retorna apenas um nodo
(será uma das folhas)
    }
    else if (iLength == 2){ //O outro caso são 2 números. Nesse caso, cria um nó
pai e um nó filho para a esquerda
        NodeTree<int>* parentNode = newNodeTree(max(vector[iStart], vector[iStart
+ 1]));
        parentNode->ptrLeft = newNodeTree(min(vector[iStart], vector[iStart +
```

```
1]));
        return parentNode;
    } //Essas operações são em tempo constante theta(1)
    // A partir de 3 entradas é possível dividir o input em esquerda, meio e
direita (semelhante ao quicksort)
    int iMedian = MOMSelect(vector, iStart, iEnd, iStart + iLength/2); //Usando
mediana das medianas pega a mediana da sequencia
    int iPivot = partitionate(vector, iStart, iEnd - 1, iMedian); //Particiona na
mediana, garantindo que para a esquerda só terão elementos menores que o pai e
para a direita elementos maiores (caracterizando uma BST)
    NodeTree<int>* parentNode = newNodeTree(vector[iPivot]); //Gera o nó pai com o
pivô
    parentNode->ptrLeft = ex15(vector, iStart, iPivot); //O filho a esquerda será
uma subárvore gerada recursivamente
    parentNode->ptrRight = ex15(vector, iPivot + 1, iEnd); //O filho a direita
será uma subárvore gerada recursivamente
    return parentNode; //Retorna o nó pai
}
```

Os casos base são resolvidos em $\theta(1)$. A operação de encontrar a mediana é realizada em tempo linear $O(n)$, bem como a operação de particionar pela mediana encontrada anteriormente (essa combinação foi usada em praticamente metade dos exercícios anteriores). Como particionamos pela mediana, sabemos que as duas funções recursivas chamadas irão receber uma entrada com tamanho $\frac{n}{2}$. A última etapa, de juntar os resultados, é em tempo constante, já que só é necessário criar os nós e ligá-los.

Dadas essas informações, podemos construir a recorrência:

$$T(n) = \begin{cases} \theta(1) & \text{se } n \leq 2 \\ 2T(\frac{n}{2}) + O(n) & \text{se } n > 2 \end{cases}$$

Para resolvê-la, podemos usar o método mestre. Como $f(n) = cn = \theta(n)$ e $n^{\log_2 2} = n$, temos que $f(n) = \theta(n^{\log_2 2})$. O teorema mestre nos diz, então, que $T(n) = \theta(n \log n)$.