## 1. Explicação das Decisões Algorítmicas e Justificativa das Classes/Métodos

**Classe `Lista`:**

- **Justificativa:** Foi criada uma classe `Lista` para encapsular a estrutura de dados baseada em uma lista sequencial do Python.
- **Método `__init__`:** O construtor inicializa a lista como vazia, conforme o requisito do projeto.
- **Método `inserir`:** A inserção foi implementada utilizando o método `append`, que adiciona elementos ao final da lista. Esta é a operação de inserção mais direta para uma lista não ordenada.
- **Método `inserirVarios`:** A inserção foi implementada utilizando o método `append`, em um for loop para adicionar vários elementos ao final da lista. Ela foi criada para evitar ter que repetir código e facilitar a leitura.
- **Método `buscar`:** O método de busca implementa uma **busca linear** (ou sequencial). Ele percorre cada elemento da lista, um a um, a partir do início, até encontrar o valor desejado ou chegar ao fim.

**Classe `Arvore`:**

- **Justificativa:** A classe `Arvore` foi implementada para representar uma Árvore Binária, uma estrutura de dados otimizada para operações de busca.
- **Método `inserir`:** A inserção segue o padrão fundamental: se o novo valor é menor que o valor de um nó, o algoritmo desce para a subárvore esquerda; se for maior, desce para a direita. O processo se repete até encontrar uma posição vazia.
- **Método `buscar`:** A busca explora um padrão para encontrar valores de forma eficiente, comparando o valor buscado com o nó atual e navegando para a esquerda ou direita, o que reduz drasticamente o espaço de busca a cada passo.

**Classe `ListaOtimizada`:**

- **Justificativa:** Foi criada uma classe ListaOtimizada com o objetivo melhorar a performance de busca de uma lista, para responder a parte 4, e é uma classe separada pra facilitar organização e consistência dos testes.
- **Método `ordenar`:** Para não ter que fazer chamadas estranhas caso houvesse somente o método do algoritmo merge sort como "lista_otimizada.items = lista_otimizada.merge_sort(lista_otimizada.items)", foi implementado a separação em dois métodos para simplesmente chamar o ordenar com Objeto.ordenar().
- **Método `merge_sort`:** Foi implementado o algoritmo de ordenação merge_sort ja que é necessário a lista estar ordenada para a busca binária, e pesquisando a diferença com quick sort, merge_sort parece mais consistente até nos piores casos.
- **Método `busca_binaria`:** Foi implementado o algoritmo de busca binária que funciona em uma lista ordenada para tentar otimizar a lista em comparação a árvore nas buscas já que funciona dividindo repetidamente o espaço de busca pela metade, e assim como a árvore tem um "guia" a seguir na sua busca, que é maior/menor.

## 2. Documentação dos Experimentos Realizados e Respostas/Conclusões

### 1: Análise do Ponto de Eficiência

Para "provar em código" a resposta, foram conduzidos múltiplos experimentos que analisaram o problema sob diferentes óticas.

Metodologia do Experimento:

Foi desenvolvida duas funções de teste que, de forma incremental, aumentava o tamanho `N` do conjunto de dados, começando com amostras pequenas e crescendo progressivamente. Para cada `N`, eram criadas uma Lista e uma Árvore, e os tempos de busca eram medidos. A análise foi dividida em dois cenários principais:

1. Eficiência de Busca Pura: medindo apenas o tempo da operação de **busca**, assumindo que as estruturas de dados já estão construídas.
2. Eficiência de Tempo Total: medindo o tempo combinado de **construção + busca**, simulando um cenário mais completo onde os dados precisam ser carregados e consultados**.**

Sendo que, para esses dois cenários citados, foi testado separadamente o ‘ponto de virada’ para a **busca de dados existentes, e dados não existentes.**

Resultados da Análise de Busca Pura:

Neste cenário, a superioridade algorítmica da Árvore é notável desde o início. Os testes revelaram que:

- Para buscas com valores **inexistentes** (o pior caso para a Lista), a Árvore se tornou mais eficiente com apenas **N = 10** elementos (o ponto de partida do teste).
- Para buscas com valores **existentes**, um cenário mais favorável à Lista, o ponto de virada ocorreu com **N = 50** elementos.

Então, se considerar somente o **tempo de busca,** a árvore tem superioridade a partir de muitos poucos valores, principalmente no caso de valores inexistentes.

Resultados da Análise de Tempo Total (Construção + Busca):

Nesse caso, como o tempo de construção da Lista é muito menor que o da Árvore, o ponto de virada aumenta um pouco. Com o conjunto de dados pequeno (começando de 100), os testes mostraram que o ponto de virada para o tempo total ocorreu em:

- **N = 120** para buscas com valores **inexistentes**.
- **N = 250** para buscas com valores **existentes**.

Isso demonstra que, para volumes de dados menores e um número de consultas proporcional ao tamanho do conjunto, a Lista mantém sua vantagem por mais tempo devido à sua construção quase instantânea.

Análise de Escalabilidade com Grandes Volumes de Dados

Para ter certeza de como seria com muitos dados e se o tempo de construção da árvore daria uma diferença grande com o da lista mesmo com a desvantagem de tempo de busca, foi realizado um teste com maiores quantidades de dados (10 mil, 50 mil, 500 mil, 1 milhão, 2 milhões…) para validar o comportamento em maior escala.

Os resultados foram conclusivos: o tempo necessário para realizar um número x de buscas na Lista tornou-se tão massivo que anulou completamente sua vantagem na construção. **A Árvore, mesmo com seu alto custo inicial, apresentou um tempo total inferior já na primeira execução do teste.**

Conclusão 1:

O ponto de virada não é um número único, mas sim dependente do cenário de uso. Em consultas rápidas sobre dados já carregados, a Árvore é superior quase que imediatamente. Se considerar o tempo de construir a estrutura para realizar buscas em volumes pequenos de dados, a Lista pode ser mais vantajosa até um máximo N de aproximadamente 250 elementos. No entanto, em qualquer cenário de larga escala, a ineficiência da busca “padrão” em uma Lista é muito grande, consolidando a Árvore como a estrutura mais escalável e eficiente imediatamente.

**2: Experimentos de Tempo**

Metodologia de Geração de Dados de Busca:

- **Valores Existentes:** Para cada conjunto de dados, foi criada uma amostra de 500 valores para busca, selecionando-se elementos em intervalos regulares a partir da lista de dados original. Isso garante que os valores buscados existem e estão distribuídos ao longo do conjunto.
- **Valores Inexistentes:** Para os testes com valores inexistentes, foi primeiro implementada uma função para encontrar o valor máximo em cada conjunto de dados. Em seguida, foi gerada uma amostra de 500 números sequenciais, começando a partir do `valor_maximo + 1`, garantindo que nenhum desses valores estaria presente na estrutura.

Resultados Brutos:

Tabela 1: Tempo de Construção das Estruturas (em segundos)

| Tamanho do Conjunto | N° de Valores | Tempo Árvore(s) | Tempo Lista(s) |
|---|---|---|---|
| Pequeno | 100.000 | 0.914971 | 0.005739 |
| Médio | 5.000.000 | 43.666933 | 0.329218 |
| Grande | 30.000.000 | 323.659801 | 2.128593 |

Tabela 2: Tempo de Busca Agregado para 500 valores (em segundos)

| Tamanho do Conjunto | Estrutura | Tempo Busca Valores Existentes | Tempo Busca Valores Não-Existentes |
|---|---|---|---|
| Pequeno | Árvore | 0.001963 | 0.000402 |
| Pequeno | Lista | 0.271309 | 1.088739 |
| Médio | Árvore | 0.001995 | 0.000565 |
| Médio | Lista | 0.270384 | 55.464146 |
| Grande | Árvore | 0.001895 | 0.000821 |
| Grande | Lista | 0.267132 | 333.250201 |

Resultados em Gráficos:

Gráfico 1: Tempo de Construção

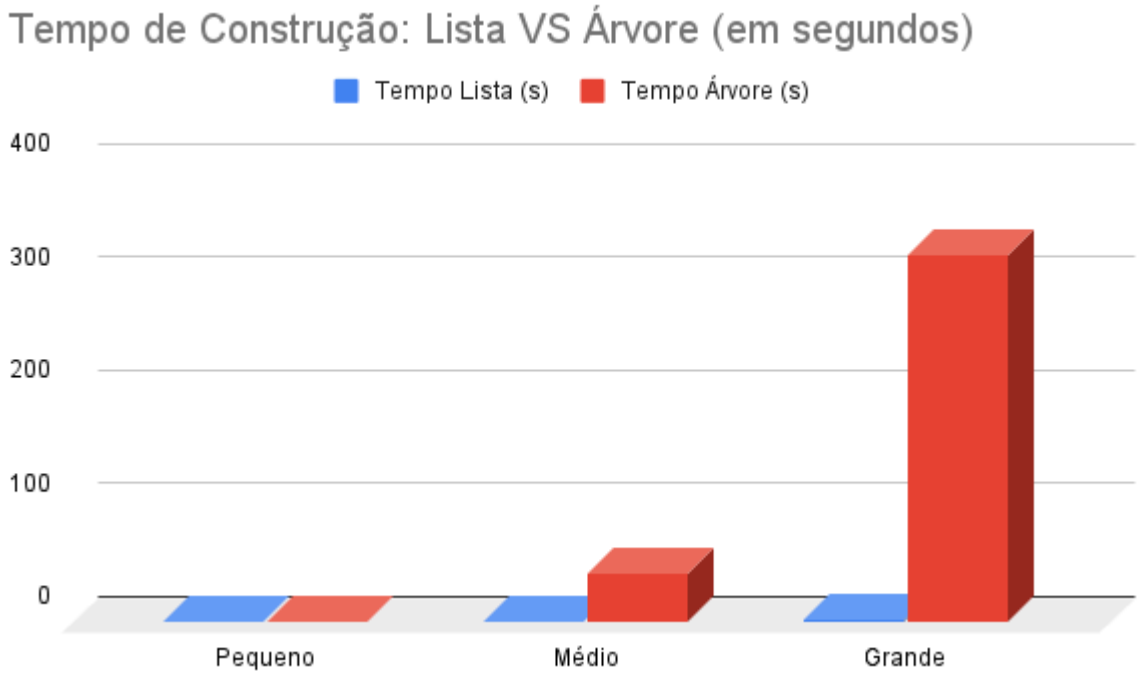
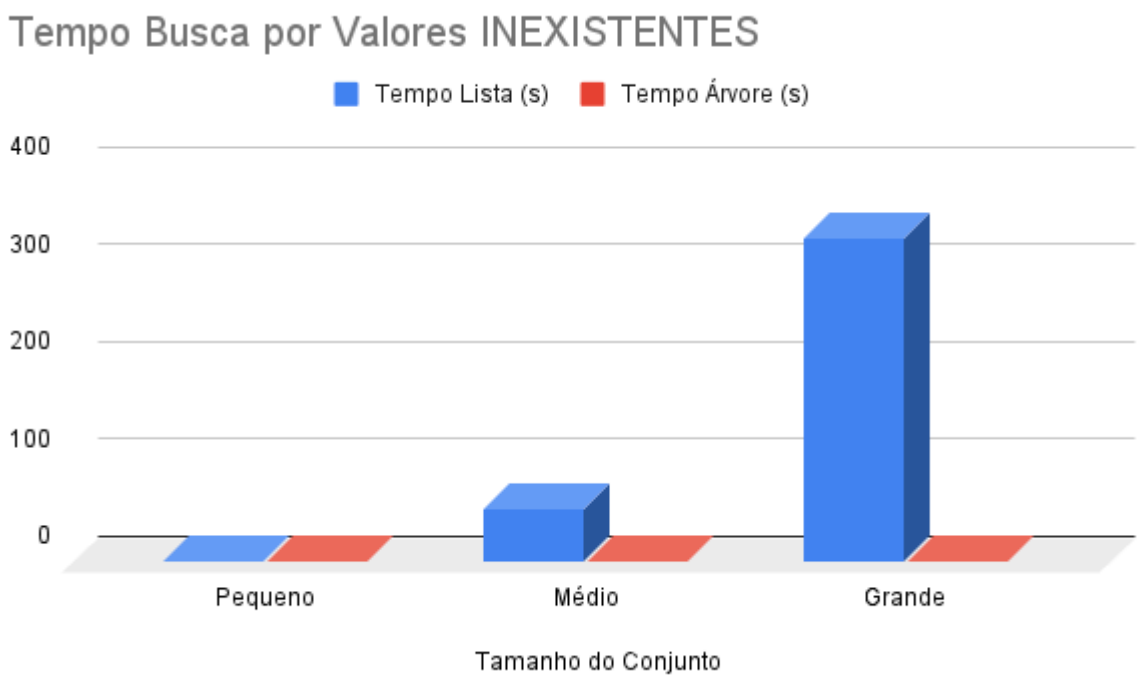



Gráfico 2: Tempo de Busca por Valores Existentes

Gráfico 3: Tempo de Busca por Valores NÃO Existentes




**Análise da Construção:** Conforme ilustrado na Gráfico 1, o tempo de construção da Lista demonstra um crescimento linear. Em contrapartida, a Árvore Binária, que requer a localização da posição correta para cada inserção, apresentou um tempo de construção significativamente maior. (outra coisa de relevância a ser citada, é que testando separadamente antes do código final para ver a diferença, a árvore consome mais memória ram que a lista)

**Análise da Busca:** A verdadeira diferença de performance é revelada nos testes de busca, para valores existentes e inexistentes existe diferença, porém especialmente para valores inexistentes é ainda pior, conforme visualizado no Gráfico 3, já que o tempo de listas passa na casa de centenas, observável no eixo vertical a esquerda. O tempo de busca na Árvore permaneceu praticamente constante e próximo de zero. Em contraste, o tempo de busca na Lista cresceu de forma linear e drástica com o aumento do volume de dados, passando de 1 segundo para mais de 333 segundos. Este comportamento evidencia a ineficiência da busca linear em grandes conjuntos de dados, que será o foco da análise na parte 3.

**3: Análise de Ineficiências**

<u>As listas se mostram mais lentas para grandes conjuntos durante a busca por valores?</u>

Sim, como demonstrado em toda a análise da parte 2, e nos gráficos e tabelas, se aumentar o número de dados, as listas têm uma discrepância de tempo de busca muito grande, e isso é ainda maior se forem dados não existentes que estão sendo buscados.

Se são ineficientes, explique detalhadamente os motivos técnicos dessa ineficiência.

As buscas em listas (ao menos essa busca 'padrão') são buscas lineares, que crescem de acordo com a quantidade de dados, isso por si só já é praticamente a resposta e também o maior motivo, mas esse cenário fica ainda pior se forem buscados dados não existentes, porque em lista, para garantir que ele não existe, a única maneira é olhando literalmente TODOS OS ITENS (5-30 milhões) até o fim para validar que 1 dos valores buscados não está presente. E para as árvores, com a sua lógica de maior/menor, você tem um "guia" de direção de onde olhar para achar o seu valor, e não precisa olhar todos os itens, mesmo se for um valor não existente.

**4: Otimização de Listas**

Para testar se tem como otimizar listas para ficarem tão eficientes quanto uma árvore binária "padrão", foi criado uma classe de "ListaOtimizada" separada que tem ordenação com o algoritmo Merge Sort e a busca é o algoritmo Busca Binária, foi testado no mesmo padrão  e com mesma exata metodologia usada na parte 2, com a diferença de ter também os dados do tempo de ordenação na etapa de construção.

Resultados Brutos:

Tabela 3: Tempo(s) de Construção da **Lista Otimizada** (em segundos)

| Tamanho do Conjunto | Tempo Construção | Tempo Ordenação | Tempo Total (construção + ordenação) |
|---|---|---|---|
| Pequeno | 0.005712 | 0.40000 | 0.405712 |
| Médio | 0.362948 | 29.201273 | 29.564220 |
| Grande | 2,073636 | 199,802485 | 201.876121 |

Tabela 4: Tempo de Busca Agregado para 500 valores para a **Lista Otimizada** (em segundos)

| Tamanho do Conjunto | Tempo Busca Valores Existentes | Tempo Busca Valores Não-Existentes |
|---|---|---|
| Pequeno | 0.002219 | 0.002041 |
| Médio | 0.004078 | 0.002679 |
| Grande | 0.004686 | 0.003169 |

**Análise da Construção:**

Uma coisa esperada que acontece, é que como agora tem que ordenar a lista para conseguir efetuar a busca binária, o tempo total de “construção” ou preparação aumenta ao somar o tempo de ordenação, no conjunto grande a lista chega em tempo total de 201s, absurdamente mais alto que antes, mas ainda abaixo do tempo de construção da árvore no conjunto grande, que foi 323s.

**Análise da Busca:**

Já **respondendo a parte 4,** como demonstrado na tabela 4 em comparação a tabela 2, é possível sim otimizar a lista sem deixar de “operar em uma lista”, a lista otimizada tem um tempo de busca muito superior a uma lista “comum”, a diferença é perceptível tanto em valores existentes, como no exemplo do conjunto grande, com a lista comum com o tempo de 0.26s, e a lista otimizada 0.004.

E assim como para outros casos no relatório, a real diferença acontece nos dados não existentes, onde a lista otimizada teve o tempo de 0.003s para o conjunto grande, quanto para a lista comum foi de absurdos 333s. Uma gigantesca diferença, já que a lista comum percorria o conjunto grande inteiro para validar cada item a ser buscado, pois não tinha nenhum algoritmo a “guiando” em uma direção, e muito menos estava ordenada. Então, mesmo se somarmos o tempo extra da ordenação necessário para lista otimizada, se mostra eficaz fazer essas implementações para o caso de realizar buscas.

**3. Análise dos Problemas Enfrentados e Soluções**

1° Problema:

- **Problema:** Durante a implementação dos testes de desempenho, foi identificado um bug bobo mas crítico no método `buscar` da classe `Lista`.
- **Análise do Bug:** O comando `return False` estava posicionado incorretamente dentro do laço `for`. Isso fazia com que a função retornasse `False` imediatamente após verificar o primeiro elemento, caso este não fosse o valor procurado, impedindo a verificação do restante da lista.
- **Solução:** O problema foi solucionado movendo o `return False` para fora e após o laço `for`, garantindo que a função só retorne `False` depois de ter percorrido todos os elementos sem sucesso.

2° Problema:

- **Problema:** Durante a implementação das funções de testes do ponto de virada, pra não ter redundância de código era usado a função de gerar dados existentes, mas ela estava implementada de uma forma fixa, dando erro de index para diferentes quantidades de dados

- **Análise do Bug:** A função gerava dados existentes usando um ‘`for`’ e pegando índices que incrementavam em um intervalo fixo, o que com diferentes quantidades dava list out of index
- **Solução:** O problema foi solucionado usando um intervalo para incrementar o índice calculado com base na quantidade fornecida para a função

3º Problema:

- **Problema:** Ao implementar o algoritmo Merge Sort, que é recursivo, surgiu uma diferença para implementar em objeto: como integrá-lo à classe `ListaOtimizada`.
- **Análise do Problema:** Uma tentativa inicial de criar um único método mergesort(self, lista) resolvia o problema da recursão (ao passar as sub-listas como parâmetro), mas a chamada do método ficava muito estranha. Para ordenar a lista, seria necessário chamar `minha_lista.mergesort(minha_lista.items)`.
- **Solução:** A solução foi refatorar a lógica utilizando um padrão que separa o algoritmo da chamada de ordenação.

**4. Configurações do Sistema usado dos testes**

**CPU: i5 10400**

**GPU: AMD RX 7600**

**RAM: 16GB**

**(armazenamento: SSD Sata 960GB Kingston)**