UFMG

# Trabalho Prático 2 (TP2)
## Algoritmos 1

**Ítalo Leal Lana Santos | Matrícula: 2024013893 | italolealanasantos@ufmg.br**
Universidade Federal de Minas Gerais (UFMG) | Belo Horizonte/MG | novembro de 2025

---

## 1. Introdução

O problema computacional proposto consiste em auxiliar o prefeito Roberto Ângulo na reforma da cidade de **Triangulândia** através de duas tarefas de otimização geométrica. A primeira envolve a reconstrução de um muro em formato de triângulo isósceles utilizando pilhas de blocos existentes, maximizando a altura sem adicionar novos blocos. A segunda requer a seleção de três árvores em um parque para delimitar uma área triangular de preservação ambiental com o menor perímetro possível.

Especificamente, o problema requer:

**Parte 1**: Determinar a altura máxima de um triângulo isósceles que pode ser construído removendo blocos de N pilhas existentes, sem adicionar novos blocos.

**Parte 2**: Identificar três árvores dentre Z árvores disponíveis em um parque que formem o triângulo de menor perímetro possível, retornando o perímetro (com quatro casas decimais) e os índices das árvores em ordem lexicográfica.

Este é um problema que combina **algoritmos gulosos** para otimização local (Parte 1) e **divisão e conquista** para geometria computacional (Parte 2). Este relatório apresenta a modelagem, solução e análise de complexidade desenvolvidas para resolver o problema proposto.

## 2. Modelagem

### 2.1 Parte 1: Construção do Muro Triangular

**Estrutura de Dados:** O problema foi modelado utilizando a classe Builder que mantém:

**stacks: list[int] -** Lista com a quantidade de blocos em cada pilha
**left: list[int] -** Altura máxima possível em cada posição considerando as pilhas à esquerda
**right: list[int] -** Altura máxima possível em cada posição considerando as pilhas à direita
**height: list[int] -** Altura efetiva do triângulo em cada posição (mínimo entre left e right)

**Modelagem do Triângulo:** Um triângulo isósceles de altura H possui sua base na posição mais baixa e os lados crescem simetricamente. Em cada posição i, a altura permitida é limitada pela distância até as bordas do triângulo. Se a posição i está a uma distância d da borda mais próxima, a altura máxima naquela posição é d.

## 2.2 Parte 2: Seleção de Árvores

**Estrutura de Dados:** O problema foi modelado utilizando duas classes:

**Point:** Representa uma árvore com índice e coordenadas (x, y)

**Grid:** Contém os pontos e métodos para encontrar o triângulo de perímetro mínimo:

- **points: dict[int, Point] -** Dicionário de pontos indexados
- **px: list[int] -** IDs dos pontos ordenados pela coordenada x
- **py: list[int] -** IDs dos pontos ordenados pela coordenada y

**Modelagem Geométrica:** O perímetro de um triângulo formado por três pontos A, B e C é a soma das distâncias euclidianas entre os pares de pontos: **P = dist(A,B) + dist(B,C) + dist(C,A)**.

# 3. Solução

### 3.1 Parte 1: Altura Máxima do Triângulo

**Algoritmo**: Programação dinâmica com abordagem gulosa
**Função**: get_maximal_height()
**Ideia Geral**: Para cada posição do muro, calculamos a altura máxima considerando as restrições do triângulo isósceles: quantidade de blocos disponíveis e geometria triangular (inclinação simétrica).
**Funcionamento do Algoritmo**:
A solução utiliza três passagens lineares:

1. **Passagem esquerda→direita**: Calcula left[i] = min(left[i-1] + 1, stacks[i]), iniciando com left[0] = 1. Isso modela o crescimento gradual da inclinação esquerda do triângulo.
2. **Passagem direita→esquerda**: Calcula right[i] = min(right[i+1] + 1, stacks[i]), iniciando com right[N-1] = 1. Modela a inclinação direita simetricamente.
3. **Cálculo final**: height[i] = min(left[i], right[i]) para cada posição, garantindo que ambas as restrições sejam satisfeitas. Retorna o máximo global.

**Estratégia Gulosa**: Escolher a maior altura possível em cada posição que satisfaça as restrições locais leva à solução ótima global.

### 3.2 Parte 2: Triângulo de Perímetro Mínimo

**Algoritmo**: Divisão e Conquista (adaptação do par de pontos mais próximos)
**Função**: divide_conquer(px_range, py_range)
**Ideia Geral**: Divide o espaço pela coordenada x, resolve recursivamente cada metade e combina verificando triângulos que cruzam a divisão. A eficiência O(Z log Z) vem de manter py_range ordenado e usar apenas filtragem linear.

**Funcionamento do Algoritmo**:

1. **Caso Base**: Para n ≤ 10, usa força bruta (brute_force_triplet()).
2. **Divisão**: Divide pelo ponto mediano em x. Cria px_left/right e py_left/right por filtragem linear, preservando ordenação por y.
3. **Conquista**: Resolve recursivamente para obter triplet_left e triplet_right.
4. **Combinação**: Compara perímetros (usa min_lexically() para empates), cria faixa (strip) por filtragem linear e verifica triângulos cruzando a divisão com find_strip_triplet().

**Otimização da Faixa**: Como o strip está ordenado por y, verifica apenas pontos próximos (diferença em y < min_perim), resultando em tempo linear O(|strip|).

**Desempate Lexicográfico**: min_lexically() compara triplas elemento por elemento, garantindo resposta única em caso de empate.

# 4. Análise de Complexidade

## 4.1 Complexidade de Tempo

**Parte 1:** Altura Máxima do Triângulo

| **Operação** | **Complexidade** | **Justificativa** |
|---|---|---|
| Passagem esquerda→direita | O(N) | Percorre N pilhas uma vez |
| Passagem direita→esquerda | O(N) | Percorre N pilhas uma vez |
| Cálculo de alturas | O(N) | Percorre N posições uma vez |

**Complexidade Total da Parte 1**: O(N) - A solução é linear pois cada pilha é processada um número constante de vezes (três passagens independentes).

**Parte 2:** Triângulo de Perímetro Mínimo

| **Operação** | **Complexidade** | **Justificativa** |
|---|---|---|
| Ordenação inicial | O(Z log Z) | Ordena pontos por x e y (uma vez) |
| Divisão e Conquista | O(Z log Z) | Recorrência com operações lineares |
| Força bruta (caso base) | O(1) | Tamanho limitado a 10 pontos |

**Análise da Recorrência**: A recorrência do algoritmo é: **T(Z) = 2T(Z/2) + O(Z)**, onde:

- **2T(Z/2)**: duas chamadas recursivas nas metades esquerda e direita
- **O(Z)**: operações de combinação realizadas em tempo linear:
  - Criação de py_left e py_right: O(Z) por filtragem linear
  - Criação do strip: O(Z) por filtragem linear
  - Verificação do strip com find_strip_triplet(): O(|strip|) ≤ O(Z)

**Justificativa do O(Z) na Combinação**: A chave para evitar O(Z log Z) no nível está em:

1. **Manter py_range ordenado**: py_range é passado já ordenado por y para as chamadas recursivas. A filtragem para criar py_left e py_right preserva essa ordenação (percorre linearmente e testa condição de x).

2. **Strip em tempo linear**: Como py_range está ordenado, criar o strip é apenas filtrar linearmente pontos próximos à linha divisória.
3. **Verificação do strip eficiente**: Para cada ponto no strip, verificamos apenas vizinhos próximos em y (limitado geometricamente), resultando em tempo linear total.

**Teorema Mestre**: Para T(Z) = 2T(Z/2) + O(Z):
- a = 2, b = 2, f(n) = O(Z)
- n^(log_b a) = Z^(log_2 2) = Z
- f(n) = Θ(Z) = Θ(n^(log_b a))

Caso 2 do Teorema Mestre: T(Z) = Θ(Z log Z)
**Complexidade Total da Parte 2**: O(Z log Z)

**Complexidade Total do Programa: O(N + Z log Z)**

## 4.2 Complexidade de Espaço

**Parte 1:** Altura Máxima do Triângulo

| Estrutura | Complexidade | Descrição |
|---|---|---|
| stacks | O(N) | Array original de pilhas |
| left, right, height | O(N) | Três arrays auxiliares |

**Espaço Total da Parte 1**: O(N)

**Parte 2:** Triângulo de Perímetro Mínimo

| Estrutura | Complexidade | Descrição |
|---|---|---|
| points | O(Z) | Dicionário de pontos |
| px, py | O(Z) | Listas ordenadas |
| Pilha de recursão | O(log Z) | Profundidade da recursão |
| Arrays temporários (px_left, etc) | O(Z) | Criados em cada nível |

**Espaço Total da Parte 2**: O(Z)

Embora arrays temporários sejam criados em cada chamada recursiva, apenas O(log Z) níveis existem simultaneamente na pilha de recursão. Como cada nível processa Z pontos no total (distribuídos entre subproblemas), o espaço extra por nível é O(Z/nível). Somando todos os níveis, temos O(Z) de espaço auxiliar.

**Complexidade Total de Espaço**: **O(N + Z)**

# 5. Considerações Finais

### 5.1 Experiência de Desenvolvimento

A Parte 1 foi relativamente direta após identificar o padrão do algoritmo guloso de programação dinâmica. A Parte 2 foi mais desafiadora, exigindo adaptação do algoritmo clássico de par de pontos mais próximos para triplas, com atenção especial à verificação da faixa e ao tratamento de empates lexicográficos.

### 5.2 Decisões de Implementação

Escolhi Python pela claridade do código. Na Parte 1, usei três arrays separados (left, right, height) priorizando legibilidade. Na Parte 2, a decisão crucial foi passar py_range já ordenado para as chamadas recursivas e usar filtragem linear para criar sublistas, mantendo a ordenação por y sem necessidade de reordenar, garantindo O(Z log Z). O caso base com limite de 10 pontos balanceou eficiência e simplicidade.

### 5.3 Desafios e Aprendizados

O maior desafio foi garantir complexidade O(Z log Z) evitando ordenações repetidas. A solução foi perceber que filtrar mantém a ordenação relativa. Outro ponto crítico foi a verificação correta do strip com limites em y adequados. O tratamento de empates lexicográficos exigiu atenção aos detalhes. Um aprendizado importante foi a adaptação de algoritmos clássicos mantendo complexidade ótima através de análise cuidadosa das operações.

## 6. Referências


CORMEN, Thomas H. et al. **Introduction to Algorithms**. 3rd ed. MIT Press, 2009. Capítulos sobre Programação Dinâmica e Divisão e Conquista.

DE BERG, Mark et al. **Computational Geometry: Algorithms and Applications**. 3rd ed. Springer, 2008. Algoritmo de par de pontos mais próximos.

KLEINBERG, Jon; TARDOS, Éva. **Algorithm Design**. Pearson, 2006. Capítulos sobre algoritmos gulosos e divisão e conquista.

Material didático da disciplina **Algoritmos I - UFMG/DCC**, 2025. Slides de aula sobre Algoritmos Gulosos e Divisão e Conquista.

Python Software Foundation. **Python 3.9+ Documentation**. Disponível em: https://docs.python.org/3/