Desafios de Programação

UMA INTRODUÇÃO A PROGRAMAÇÃO COMPETITIVA

por Fernando Kiotheka, Raul Almeida e Vinicius Tikara Date

compilado em 22 de julho de 2025

2023

Sumário

Introdução

O objetivo principal desse livro é dar suporte à matéria de Desafios de Programação ministrada no curso de Bacharelado em Ciência da Computação da Universidade Federal do Paraná. Queremos por meio dessa matéria, principalmente, fortalecer o grupo de maratona da UFPR (chamado Capimara UFPR).

Porém, esse tipo de conhecimento não é restrito apenas a programação competitiva. É comum, por exemplo, empresas usarem no seu processo de entrevista o emprego de desafios lógicos ou desafios de caráter similar. Mas além disso, muitos dos algoritmos explorados aqui servem para resolver problemas do mundo real, e mesmo se eles não resolvessem, algoritmos são divertidos.

É esperado que o leitor tenha familiaridade com alguns básicos de lógica de programação.

Capítulo 1

C++, Juízes e Complexidade

Para resolver problemas, precisamos estabelecer uma linguagem de programação que não nos limita em capacidade, e que seja rápida. Também é útil que muitas coisas já estejam prontas, para evitar ficar digitando o mesmo código. Porém, também temos que estar atentos que o conjunto de linguagens que são aceitos em competições geralmente é extremamente restrito.

Então, para a nossa jornada no mundo da programação competitiva, vamos focar na linguagem C++, e mais especificamente, na versão de 2020 que no momento da digitação deste livro é a que está disponível nas competições. Outras linguagens são dignas de menção: Java e Python. Elas por vezes são mais expressivas, mas tem problemas com verbosidade e velocidade de execução.

A linguagem C++ é uma extensão a linguagem C e tem uma porção de baterias incluídas, ou seja, possui muitas estruturas de dados e algoritmos já prontos. E ainda por cima, a cada nova versão, mais funcionalidades são integradas a sua biblioteca padrão. Podemos citar por exemplo:

- Alocação dinâmica (vetores que crescem de tamanho sozinhos)
- Árvores balanceadas (geralmente rubro-negras)
- Filas de prioridade
- Ordenação
- Pares (ordenação já embutida)
- Números complexos (pontos 2D)
- Geração de permutações
- Geração de números aleatórios
- Expressões regulares
- Manipulação de strings de forma dinâmica
- Entrada/Saída mais concisa (em comparação ao C)

Vale notar que usaremos o C++ com o propósito de programação competitiva. Isso significa que o código não vai seguir boas práticas de desenvolvimento de software. Abusaremos de globais e do operador vírgula, poluíremos o espaço de nomes principal com a biblioteca padrão (incluíndo absolutamente todas as bibliotecas existentes), e vamos configurar a entrada e saída para serem mais rápidas (ficando "impróprias" para humanos).

Como ponto de partida, vamos ver uma estrutura básica de um programa em C++ para programação competitiva:

Código `estrutura-basica.cpp`

Incluímos todos os cabeçalhos da STL de uma vez só (equivalente a `#include <set>`, `#include <vector>`, etc). Essa funcionalidade é a princípio, **exclusiva ao GCC** mas você pode criar o seu próprio cabeçalho se for necessário. Outro problema é que fazer isso aumenta consideravelmente o tempo de compilação, principalmente quando você acabou de ligar o computador.

```
#include <bits/stdc++.h>
```

Usamos o espaço de nomes `std` (biblioteca padrão). Assim, para usar um `vector` não precisamos digitar `std::vector`.

```
using namespace std;
```

Encurtamos o `long long` que é bastante usado

```
using ll = long long;
```

Função `main` padrão. É possível usar `signed` (que é equivalente a `int`) ao invés de `int` para permitir barbaridades como `#define int long long`. Vale lembrar que conforme o padrão do C++ Seção 3.6.1, item 2 (p. 58)[1], a função `main` precisa ser do tipo `int`.

```
int main() {
```

Desvinculamos a entrada da saída padrão. O comportamento padrão é fazer um `flush` da saída padrão ao ler da entrada. Isso é mais amigável a seres humanos que esperam um programa em linha de comando dizer que eles precisam digitar alguma coisa, mas também faz com que a saída fique bem mais lenta do que gostaríamos. Então, ao desvincular, reduzimos os `flush`s a apenas quando o buffer de saída estiver cheio, o que é mais rápido.

```
    cin.tie(0);
```

Dessincronizamos as funções do `<stdio.h>` (`printf`/`scanf`/`fprintf`) com as do C++ (`cin`/`cout`/`cerr`). Isso significa que você **não pode misturar** funções da `<stdio.h>` com as do C++ no mesmo canal (`scanf`+`cout` é válido, `printf`+`cout` não é).

```
    ios_base::sync_with_stdio(0);
```

```
    // implemente sua lógica
```

O `return 0` é implícito, como definido no padrão do C++ Seção 3.6.1, item 5 (p. 59)[1] (e isto é uma peculiaridade da função `main`)

```
}
```

[1] Working Draft, Standard for Programming Language C++. (C++11) http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf

Em todo este livro, vamos utilizar essa estrutura básica, pulando por brevidade as duas linhas de entrada/saída rápida (mas que são de suma importância para resolver problemas de programação competitiva). De forma didática, mostraremos programas completos que resolvem problemas com limites específicos para ajudar o leitor a ter um ponto de partida.

Porém, não basta apenas abrir o editor de texto e codificar um problema. É boa prática que também testemos o programa no nosso próprio ambiente de trabalho antes de submetê-lo para qualquer lugar que seja. Você deve ter notado que usamos por exemplo uma funcionalidade específica ao GCC na nossa estrutura básica, porque este será o compilador que será usado como base para este livro.

Se fizermos um código portável, importando manualmente cada *header*, podemos utilizar tanto o GCC (executável `g++`) quanto o Clang (executável `clang++`). Os pacotes de ambos os compiladores são facilmente encontrados em qualquer distribuição de Linux, e escolher qual binário o atalho `c++` aponta nas distribuições Debian é feito pelo `update-alternatives(1)`. Por exemplo, podemos compilar sem dar um nome para a aplicação (sendo o padrão, `a.out`), ou dando um nome, e em seguida, rodar:

```
$ c++ codigofonte.cpp -O2 -o programa && ./programa
$ c++ codigofonte.cpp -O2 && ./a.out
```

Outra opção que é popular é utilizar o Make para compilar nossos arquivos. O Make possui regras implícitas de compilação que nos agilizam muito o trabalho, e a criação de um Makefile nos permite também modificar as regras implícitas com mais `flags` de compilação. A regra implícita que nos interessa é aquela que converte arquivos `%.cpp` em arquivos binários `%`:

```
$ echo "CPPFLAGS=-O2" >Makefile
$ make programa && ./programa
g++ -O2 programa.cpp -o programa
```

É extremamente recomendado compilar com o *flag* de otimização `-O2` para ter resultados mais próximos aos do juiz em velocidade e erros de execução. Ele irá revelar alguns alertas que o compilador normalmente não daria em seu modo padrão e principalmente, vai otimizar o programa em diversos lugares fazendo com que a sua estimativa de tempo tomado pelo programa seja mais fiel a do juiz.

Vale lembrar que o compilador toma a liberdade de abusar de quaisquer comportamentos indefinidos que estão no padrão. Isso significa que por exemplo, você não pode ficar triste quando você esquecer de retornar um valor em uma função (que não seja o `main`) e levar uma falha de segmentação quando compilar com `-O2` mas não quando compilar sem nada.

Outros *flags* de compilação que são bem úteis são:

`-g` Permite depuração e melhora o uso do `valgrind` (uma ferramenta extremamente útil para encontrar exatamente o lugar onde você acessou memória inválida).

`-fsanitize=undefined,address,memory` No momento da execução, aborta o programa em caso de comportamento indefinido, acesso fora dos limites dos vetores ou acesso a memória inválida (`memory` é exclusivo do Clang).

`-static` Geralmente usado nos juízes, permite que o tempo de execução do executável seja mais consistente, pois elimina o tempo utilizado para fazer ligação dinâmica durante a execução do programa.

Capítulo 2

Estruturas Padrão

Como citado anteriormente, um dos motivos de se utilizar o C++ como linguagem se deve à miríade de estruturas que a STL nos proporciona. Neste capítulo iremos abordar as mais utilizadas em programação competitiva, e algumas de suas funções/métodos principais.

2.1 O clássico vetor (`vector`)

O `vector<T>` é um vetor: Uma sequência contígua de memória alocada na *heap* do tipo `T` especificado que aumenta de tamanho sozinha, podendo ser endereçada por índices aleatórios em $\mathcal{O}(1)$.

- `vector<T> v (int n = 0, T val = {})` [$\mathcal{O}(n)$]: Cria um `vector` do tipo `T` com `n` elementos inicializados em `val`.
- `v.resize(int n, T val = {})` [$\mathcal{O}(n)$]: Muda o tamanho do vetor preenchendo com `val` se `n` for maior que o atual.
- `v.size()` [$\mathcal{O}(1)$]: Retorna o tamanho (cuidado: `unsigned`!).
- `v[size_t i]` [$\mathcal{O}(1)$]: Acessa o índice i.
- `v.front()`, `v.back()` [$\mathcal{O}(1)$]: Acessa o primeiro, último.
- `v.push_back(T x)` [$\mathcal{O}(1)$]: Adiciona um valor ao final.
- `v.pop_back()` [$\mathcal{O}(1)$]: Remove o elemento do final.
- `v.clear()` [$\mathcal{O}(1)$]: Remove todos os elementos.
- `v.empty()` [$\mathcal{O}(1)$]: Está vazio?

2.2 O par ordenado (`pair`)

O `pair<T, U>` é um par ordenado é composto de dois elementos do tipo `T` e `U` respectivamente. Ele é extremamente útil porque você não precisa criar uma struct quando precisar de um par de elementos, e além disso, ele é ordenável, por padrão pelo primeiro campo.

- `pair<T, U> p (T x, U y)`: Cria um par com x e y.
- `p.first`: Acessa o primeiro item do par.
- `p.second`: Acessa o segundo item do par.
- `make_pair(T x, U y)`: Cria um par com x e y.

2.3 A tupla (`tuple`)

Em algumas situações, o `tuple<T u, U y, ...>` é necessário, quando temos que utilizar mais do que um par para a resolução do problema. Da mesma forma que o par, a tupla também é ordenável, começando do primeiro campo dela em diante.

- `tuple<T, U, ...> t (T x, U y, ...)`: Declara uma tupla com x, y, ...
- `make_tuple(T x, U y, ...)`: Cria uma tupla com x, y, ...
- `get<i>`: Acessa a posição i da tupla ou par.

2.4 A fila de duas extremidades (`deque`)

É similar ao `vector`, com a diferença de que é possível inserir e retirar itens do começo da estrutura. A contingência de memória não é garantida, o que pode vir a ocasionar constantes maiores com seu uso.

- `push_front(T x)`: Insere um item no começo da `deque`.
- `pop_front(T x)`: Remove o item do começo da `deque`.

2.5 A pilha (`stack`)

Estrutura de pilha simples LIFO (**L**ast **I**n **F**irst **O**ut), podemos apenas inserir e remover do topo da estrutura. Em sua implementação base é utilizada a `deque`, mas se prefirir você pode mudar para uma `list` ou um `vector`: `stack<T, vector<T>>`.

- `stack<T>()`: Cria a pilha.
- `empty()`: Verifica se a pilha está vazia.
- `push(T x)`: Insere um item x no topo da pilha.
- `pop()`: Remove o topo da pilha.
- `top()`: Retorna uma referência do topo da pilha.
- `size()`: Retorna o tamanho da pilha.

2.6 A fila (`queue`)

Fila simples, abstração em cima da `deque`, efetivamente limitando apenas o uso frontal. O primeiro item que entra é o primeiro que sai, ou seja, FIFO (**F**irst **I**n **F**irst **O**ut).

- `queue<T>`: Cria a fila.
- `push(T x)`: Insere um item x na frente da fila.
- `pop()`: Remove a frente da fila.
- `front()`: Retorna uma referência da frente da fila.

2.7 A fila de prioridades (`priority_queue`)

Também conhecida como `heap`. É uma fila que ordena os membros automaticamente ao serem inseridos. A base é o `vector`. Sua função de comparação base é de máximo, os itens de maior valor vão para frente da fila. Podemos definir uma função de comparação customizada para a estrutura e utilizar no construtor.

Tem as mesmas funções de uma fila normal.

- `priority_queue<T>`: Cria a fila de prioridades.
- `priority_queue<T, vector<T>, comp>`: Cria a fila de prioridades com comparador próprio.
- `push(T x)`: Insere um item x na fila.
- `pop()`: Remove a frente da fila.
- `top()`: Retorna uma referência da frente da fila.

2.8 A lista (`list`)

Implementação de uma lista duplamente encadeada, sua memória não é contígua. A lista não permite o acesso aleatório, operações na lista necessitam que seja iterado até posição desejada. Uma vez que temos o iterador, a operação de inserção/deleção tem custo constante.

Pelo fato de ser uma lista ligada, é possível concatenar elementos de uma lista em outra, utilizando o *splice*.

- `list<T>`: Cria a lista.
- `insert (iterator position, T x)`: Insere valor x na posição anterior do iterador.

2.9 O conjunto (`set`)

Estrutura que representa um conjunto matemático, a estrutura não permite a repetição de elementos, mesmo se múltiplos de um valor forem inseridos.

Na sua implementação é utilizada uma árvore binária balanceada, de tal forma que a inserção e deleção são $\mathcal{O}(lg(n))$.

É possível iterar sobre o conjunto, pois existe uma ordem na estrutura de implementação, entretanto também não temos o acesso aleatório.

- `set<T> s`: Cria um conjunto de tipo T.
- `find(T value)`: Método que procura pelo valor no conjunto, retornando um iterador para o fim, após o último elemento, se não achar.

2.10 O dicionario (`map`)

Dicionário simples de chave-valor, com ambos tipos sendo definidos pelo programador. Como o conjunto, sua implementação é feita através de uma árvore balanceada, utilizando pares, portanto, igual o conjunto, tem complexidade $\mathcal{O}(lg(n))$.

Muito utilizado para a compressão de coordenadas(Não é possível ter um range de 10^9 elementos afinal, a memória dos problemas não permite).

- `map<T, U> m`: Cria um dicionário indexado por chaves do tipo T e com valores do tipo U.

2.11 O conjunto com elementos repetidos (`multiset`)

O `multiset` ...

2.12 (unordered_...)

Em alguns problemas a complexidade logarítmica das estruturas `map` e `set` não é o suficiente, necessitamos de uma complexidade menor, no caso **constante**.

Para obter ela podemos declarar ambas estruturas como `unordered`, que indica que ao invés de uma árvore em sua implementação, é usada uma hash table, isto é, é aplicada uma função à chave, que então retorna a posição do valor na tabela em que é armazenada.

Entretanto, existem problemas decorrentes desta abordagem, a implementação padrão é conhecida e os problemsetters experientes conseguem fazer com que a hash tenha muitas colisões. Isto gera um aumento do custo, a complexidade não é mais constante, pode ser que até vire linear!(Pior do que sua contra parte ordenada). Um dos caminhos para evitar isso é utilizar funções de hash próprias.

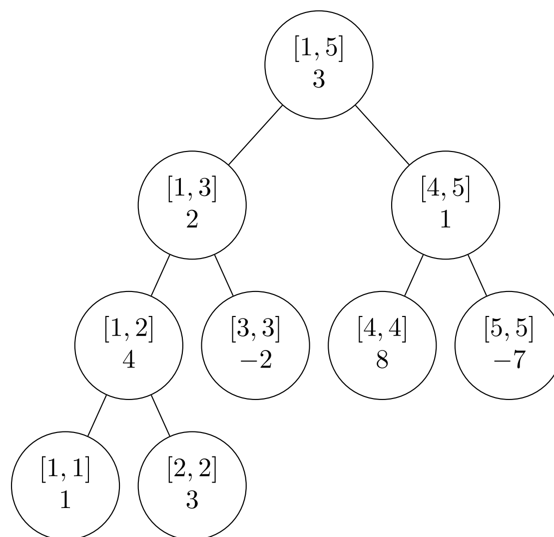
- `unordered_map<T, U> m`: Cria um `map` não ordenado.
- `unordered_set<T> s`: Cria um `set` não ordenado.

Capítulo 3

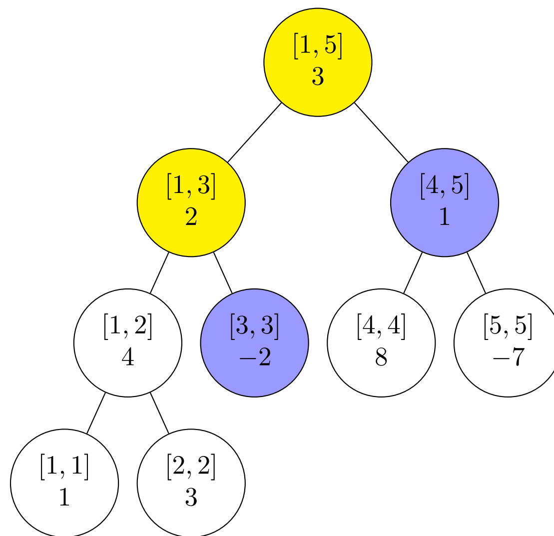
Árvore de Segmentos

A Árvore de Segmentos (Segment Tree, SegTree) é uma árvore binária onde cada nodo corresponde a um intervalo no seu vetor de consulta e atualização. Os nodos filhos correspondem às duas metades do intervalo.

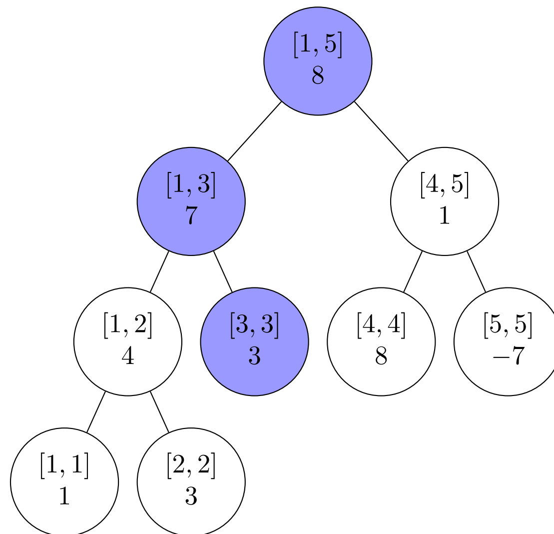
Para o vetor $V = [1, 3, -2, 8, -7]$:



Qual a soma do intervalo $[3, 5]$?



Atribuindo o valor 3 ao elemento de índice 3:



A implementação recursiva é mais fácil de se entender e de memorizar, porém ela exige ao menos $4N$ de memória.

3.1 Árvore de Segmentos Preguiçosa