Técnicas para Programação Competitiva

Técnicas de Programação

Samuel da Silva Feitosa

Aula 2
2022/2

Técnicas de Programação

Features da Linguagem C++

- Um template de código típico de programação competitiva será algo como:

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

using namespace std;

int main() {
    // solution comes here
}
```

- <bits/stdc++.h> inclui toda a biblioteca padrão.
- ***using namespace std*** é usado para não precisar informar o namespace para cada chamada da biblioteca padrão.

Compilando o código...

- O código desenvolvido pode ser compilado com o seguinte comando:

```
g++ -std=c++11 -O2 -Wall test.cpp -o test
```

- Este comando produz um binário chamado *test* a partir do código *test.cpp*.
  - Neste comando estamos indicando para usar o compilador C++11 padrão, otimizar o código com -O2 e mostrar *warnings* sobre possíveis erros.

Entrada e Saída (1)

- Na maioria das competições, é utilizada a entrada e saída padrão do sistema.
  - Em C++ usamos *cin* para entrada e *cout* para saída.
  - Também é possível utilizar as funções *scanf* e *printf* do C.
- A entrada geralmente consiste de números e strings separadas por espaços e quebras de linhas, as quais podem ser lidas usando *cin*. Para a saída usamos o *cout*.

```
int a, b;
string x;
cin >> a >> b >> x;
```

```
int a = 123, b = 456;
string x = "monkey";
cout << a << " " << b << " " << x << "\n";
```

Entrada e Saída (2)

- Algumas vezes, a entrada e saída são gargalos do programa.
    - As linhas seguintes no começo do código trazem mais eficiência.
    - Prefira usar “\n” ao final da linha, ao invés de *endl*, pois o *endl* sempre faz flush.

```
ios::sync_with_stdio(0);
cin.tie(0);
```

Trabalhando com números

- **Integers**
  - ***int*** vai de -2^{31} à $2^{31} - 1$ (aprox. $-2 * 10^9$ e $2 * 10^9$).
  - ***long long*** vai de -2^{63} à $2^{63} - 1$ (aprox. $-9 * 10^{18}$ e $9 * 10^{18}$).
- Tomar cuidado ao misturar diferentes tipos.
  - O código abaixo, apesar de atribuir o resultado em um ***long long*** é calculado a partir de um ***int***, produzindo um ***int*** com valor incorreto.

```
int a = 123456789;
long long b = a*a;
cout << b << "\n"; // -1757895751
```

Aritmética Modular

- Algumas vezes, a resposta de um problema é um número muito grande, sendo solicitado a impressão do resultado como “módulo *m*”.
  - Isto é, o resto da divisão por *m* (ex.: “módulo $10^9 + 7$”).
  - Em C++ usamos o operador % para representar o módulo.
- Vejamos algumas propriedades:

$$(a + b) \bmod m = (a \bmod m + b \bmod m) \bmod m$$
$$(a - b) \bmod m = (a \bmod m - b \bmod m) \bmod m$$
$$(a \cdot b) \bmod m = (a \bmod m \cdot b \bmod m) \bmod m$$

Complexidade de Tempo

Complexidade de Tempo

- A eficiência de algoritmos é importante na Programação Competitiva.
  - Geralmente é fácil projetar um algoritmo que resolve o problema, porém, ele não é capaz de passar no tempo esperado pelas plataformas de julgamento.
  - O desafio é projetar um algoritmo que seja eficiente.
- A complexidade de tempo de um algoritmo estima quanto tempo que o algoritmo vai levar para processar alguma entrada.
  - A ideia é representar a eficiência como uma função, a qual possui como parâmetro o tamanho da entrada.
  - Calculando (ou estimando) a complexidade é possível descobrir se o algoritmo é rápido o suficiente sem implementá-lo.

Regras de Cálculo (1)

- A complexidade de tempo (pior caso) de um algoritmo é denotado por O(...), onde '...' representa alguma função.
    - Geralmente a variável *n* denota o tamanho da entrada.
- Se o código consiste de comandos únicos, a complexidade é O(1).

```
a++;
b++;
c = a+b;
```

- A complexidade de um loop estima o número de vezes que o código dentro do loop é executado. A complexidade do loop a seguir é O(n):

```
for (int i = 1; i <= n; i++) {
    ...
}
```

Regras de Cálculo (2)

- A complexidade do loop a seguir é $O(n^2)$:

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        ...
    }
}
```

- Em geral, se existem *k* loops aninhados e cada loop percorre *n* valores, a complexidade do loop é $O(n^k)$

Regras de Cálculo (3)

- A complexidade de um loop não nos diz exatamente quantas vezes o código interno é executado
- A complexidade nos dá apenas uma ordem de crescimento, ignorando fatores constantes
- No loop abaixo, a quantidade exata de vezes que o código interno é executado é 3*n*, mas a sua complexidade é O(n)

```
for (int i = 1; i <= 3*n; i++) {
    ...
}
```

Regras de Cálculo (4)

- A complexidade do loop a seguir é $O(n^2)$:

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= i; j++) {
        ...
    }
}
```

- Esta complexidade pode ser calculada como $1 + 2 + \ldots + n = ½ * (n^2 + n)$

Regras de Cálculo (5)

- Se um algoritmo consiste de várias fases consecutivas, a complexidade do algoritmo é dada pela maior entre as complexidades das fases
- A complexidade do algoritmo a seguir é $O(n^2)$:

```
for (int i = 1; i <= n; i++) {
    ...
}
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        ...
    }
}
for (int i = 1; i <= n; i++) {
    ...
}
```

Complexidades de Tempo Comuns (1)

- $O(1)$, representa tempo constante.
- $O(\log n)$, complexidade logarítmica, geralmente divide na metade a entrada a cada passo de execução.
- $O(n)$, representa um algoritmo linear.
- $O(n \log n)$, esta complexidade geralmente indica que o algoritmo ordena a entrada.
- $O(n^2)$, um algoritmo quadrático, geralmente possui dois loops aninhados.
- $O(n^3)$, um algoritmo cúbico, geralmente possui três loops aninhados.

Complexidades de Tempo Comuns (2)

- Há ainda as complexidades $O(2^n)$ e $O(n!)$ que geralmente indicam algoritmos não eficientes.
- Um algoritmo é dito **polinomial** se a sua complexidade de tempo é no máximo $O(n^k)$, onde k é uma constante.
- A maioria dos problemas que vamos estudar são de tempo polinomial.
  - Entretanto, existe toda uma classe de problemas que não se sabe se existe solução eficiente (polinomial), os quais estão relacionados à classe NP.

Estimando a Eficiência (1)

- Por calcular a complexidade de tempo de um algoritmo, é possível verificar, antes de implementar, que um algoritmo é eficiente ou não.
  - Sabe-se que um computador moderno é capaz de executar centenas de milhões de operações simples em um segundo.
- Por exemplo, vamos assumir que o tempo limite de um problema é 1 segundo, e o tamanho da entrada é $n = 10^5$.
  - Se o algoritmo é $O(n^2)$, ele vai executar aproximadamente $(10^5)^2 = 10^{10}$ operações. Isto deve tomar pelo menos algumas dezenas de segundos, o que indica que o algoritmo é muito lento para resolver o problema.
  - Entretanto, se a complexidade é $O(n \log n)$, serão $10^5 \log 10^5 \approx 1.6 * 10^6$ operações, e o algoritmo deve executar no tempo desejado.

Estimando a Eficiência (2)

- Por outro lado, é possível *adivinhar* a complexidade de tempo necessária de um algoritmo dado o tamanho da entrada.

Input size	Expected time complexity
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(n \log n)$ or $O(n)$
n is large	$O(1)$ or $O(\log n)$

- Por exemplo, se o tamanho da entrada é $n = 10^5$, provavelmente uma complexidade O(n) ou O(n log n) deve ser suficiente.

Considerações Finais

- Nesta aula iniciamos nossos estudos com técnicas de programação em C++ aplicadas ao contexto da programação competitiva.
- Também vimos como estimar o tempo de execução de um algoritmo, e como isso pode ser útil para identificar o tipo de solução para um determinado problema.

Exercícios Ad-hoc

- Resolver pelo menos 3 exercícios introdutórios da plataforma CSES.
    - Dois estudantes serão sorteados para apresentar a solução de uma das questões resolvidas no início da próxima aula.