fusberti@ic.unicamp.br

Desafios de Programação

prof. Fábio Luiz Usberti

MC521 - Desafios de Programação

Instituto de Computação - UNICAMP - 2015

Sumário

Programação competitiva

Conceito

Programação competitiva consiste em resolver problemas conhecidos de computação sob restrições de tempo de implementação, tempo de execução e memória.

- Problemas conhecidos: em programação competitiva, os problemas são conhecidos e já foram resolvidos pela comunidade acadêmica; portanto, não se tratam de problemas que estão na fronteira da ciência.
- Tempo de implementação: trata-se de um dos elementos competitivos, presentes nas competições de programação.
- Tempo de execução e uso de memória: restrições frequentemente presentes nos problemas de programação competitiva.

Tornando-se um programador competitivo

Dica 1: Desenvolver habilidade em digitação

- Quando dois competidores conseguem resolver o mesmo número de problemas, o mais competitivo será aquele que possuir a melhor habilidade em programar (códigos concisos e robustos) e digitar.
- É comum observar em competições de programação times cujas classificações estão separadas por apenas alguns minutos.
- Alguns competidores podem perder pontos importantes porque não conseguiram implementar uma solução de força-bruta nos últimos minutos.
- Teste de digitação: http://www.typingtest.com
- Familiarize-se com as posições dos caracteres utilizados com frequência em linguagens de programação, como
  `() [] <> ; : ' " & | !`

Tornando-se um programador competitivo

Dica 2: Identificar o tipo de problema

- Determinar a qual categoria o problema melhor se enquadra (vide tabela abaixo).

No	Category	In This Book	Frequency
1.	Ad Hoc	Section 1.4	1-2
2.	Complete Search (Iterative/Recursive)	Section 3.2	1-2
3.	Divide and Conquer	Section 3.3	0-1
4.	Greedy (usually the original ones)	Section 3.4	0-1
5.	Dynamic Programming (usually the original ones)	Section 3.5	1-3
6.	Graph	Chapter 4	1-2
7.	Mathematics	Chapter 5	1-2
8.	String Processing	Chapter 6	1
9.	Computational Geometry	Chapter 7	1
10.	Some Harder/Rare Problems	Chapter 8-9	1-2
		Total in Set	8-17 ($\approx \leq 12$)

Fonte: Halim e Halim, 2013.

Tornando-se um programador competitivo

Dica 2: Identificar o tipo de problema

- Para ser competitivo, um programador deve ser capaz de classificar um problema em uma categoria para a qual ele já resolveu muitos outros problemas, o que o tornará capaz de resolvê-lo em pouco tempo.
- Para adquirir essa habilidade é necessário que o programador adquira um bom volume e diversidade de problemas resolvidos. Essa cultura é obtida com esforço e auto-didatismo.
- Para ser bem-sucedido nos desafios de programação, também é necessário desenvolver proficiência na resolução de novos problemas, que “parecem” não se enquadrar em nenhuma categoria. Em geral, isso envolve reduzir o novo problema a um problema conhecido, identificar características e propriedades que facilitam o problema, atacar o problema por uma abordagem diferenciada).

Tornando-se um programador competitivo

Dica 3: Fazer análise de algoritmos

- Normalmente há diversas formas de resolver um mesmo problema, mas muitas delas podem não ser rápidas o suficiente, enquanto outras podem ser como “canhões para matar uma mosca”.
- Uma vez definido o algoritmo a ser implementado para resolver um problema, devemos nos perguntar se a maior entrada possível (geralmente descrita no enunciado) vai ser resolvida considerando as restrições de tempo de execução e memória utilizada.
- Uma boa estratégia consiste em escolher o algoritmo mais simples que resolva o problema, respeitando as restrições impostas. É comum o algoritmo mais simples ser o menos eficiente, mas se ele atende a restrição de tempo de execução, não há porque não utilizá-lo.

Dica 3: Fazer análise de algoritmos

Regras gerais para análise de tempo de algoritmos iterativos e recursivos:

- Um algoritmo com k laços aninhados, cada um com n iterações, terá complexidade $O(n^k)$.
- Um algoritmo recursivo com b chamadas recursivas em cada um dos L níveis, terá uma complexidade $O(b^L)$ (na verdade pode ser muito melhor do que isso).
- Um algoritmo de programação dinâmica, ou outra rotina iterativa, que processa uma matriz $n \times n$, consumindo um tempo $O(k)$ em cada célula, terá complexidade $O(kn^2)$.

Essas regras estão longe de esgotar o assunto de complexidade de algoritmos. Para uma visão mais detalhada, consultar as referências bibliográficas da disciplina.

Tornando-se um programador competitivo

Dica 3: Fazer análise de algoritmos

n	Worst AC Algorithm	Comment
$\leq [10..11]$	$O(n!), O(n^6)$	e.g. Enumerating permutations (Section 3.2)
$\leq [15..18]$	$O(2^n \times n^2)$	e.g. DP TSP (Section 3.5.2)
$\leq [18..22]$	$O(2^n \times n)$	e.g. DP with bitmask technique (Section 8.3.1)
≤ 100	$O(n^4)$	e.g. DP with 3 dimensions + $O(n)$ loop, ${}_nC_{k=4}$
≤ 400	$O(n^3)$	e.g. Floyd Warshall's (Section 4.5)
$\leq 2K$	$O(n^2 \log_2 n)$	e.g. 2-nested loops + a tree-related DS (Section 2.3)
$\leq 10K$	$O(n^2)$	e.g. Bubble/Selection/Insertion Sort (Section 2.2)
$\leq 1M$	$O(n \log_2 n)$	e.g. Merge Sort, building Segment Tree (Section 2.3)
$\leq 100M$	$O(n), O(\log_2 n), O(1)$	Most contest problem has $n \leq 1M$ (I/O bottleneck)

Fonte: Halim e Halim, 2013.

Tornando-se um programador competitivo

Dica 3: Fazer análise de algoritmos

- Computadores pessoais modernos executam na ordem de 30 milhões de instruções por segundo (3×10^7) (*Intel Core i7 3770k*). É possível utilizar essa informação para avaliar se um algoritmo pode terminar antes do limite de tempo de execução.
- Exemplo: Suponha um algoritmo de complexidade $O(n \log_2 n)$ para um problema com entrada máxima de tamanho $n = 10^3$. Sabendo que $10^3 \log_2 10^3 \approx 10^4$, é possível presumir com alguma segurança que o algoritmo deverá executar em menos de um segundo.

Dica 3: Fazer análise de algoritmos

Informações úteis:

- $2^{10} = 1.024 \approx 10^3$, $2^{20} = 1.048.576 \approx 10^6$
- o tipo `int` (C++, Java) ocupa 32 bits e pode representar até o número $2^{31} - 1 \approx 2 \times 10^9$.
- o tipo `long long` (C++; somente `long` em Java) ocupa 64 bits e pode representar até o número $2^{63} - 1 \approx 9 \times 10^{18}$.
- Há $n!$ permutações e 2^n subconjuntos de n elementos.
- Algoritmos com complexidade $O(n \log_2 n)$ são suficientes para passar por qualquer problema em programação competitiva (exceto raras exceções).
- A maior entrada de um problema em competições é raramente maior do que 1.000.000.
- Os computadores pessoais atuais executam na ordem de 30 milhões de instruções (3×10^7) por segundo (*Intel Core i7 3770k*).

Tornando-se um programador competitivo

Dica 4: Tornar-se fluente em linguagens de programação

- Linguagens de programação permitidas em competições oficiais incluem C, C++ e Java.
- Um programa Java é, em geral, considerado mais lento do que um programa em C++. Por outro lado:
  - Um código em Java é mais fácil de ser depurado, dado que a máquina virtual providencia toda a pilha de execução após uma exceção.
  - A linguagem Java possui um conjunto de APIs como `Regex`, `BigInteger`, `GregoriaCalendar`, dentre outras que podem ser muito úteis para certos tipos de problema.

Tornando-se um programador competitivo

Dica 4: Tornar-se fluente em linguagens de programação

Exemplo: problema cuja solução em Java torna-se muito vantajosa.

- Escreva um programa que calcule o fatorial de 25.
- $25! = 15.511.210.043.330.985.984.000.000$

```
import java.util.Scanner;
import java.math.BigInteger;

class Main {     // standard Java class name in UVa OJ
  public static void main(String[] args) {
    BigInteger fac = BigInteger.ONE;
    for (int i = 2; i <= 25; i++)
      fac = fac.multiply(BigInteger.valueOf(i));
    System.out.println(fac);
  }
}
```

Tornando-se um programador competitivo

Dica 5: Aperfeiçoar sua habilidade em depurar códigos

- Ao submeter um código a um corretor automático, desejamos receber como veredito: Accepted (AC).
- Outros possíveis vereditos (indesejáveis) são:
  1. Presentation Error (PE) – a saída do seu programa está correta, porém não está formatada como especificado no enunciado.
  2. Wrong Answer (WA) – a saída do seu programa está incorreta em pelo menos um dos casos de teste.
  3. Time Limit Exceeded (TLE) – o seu programa esgotou a restrição de tempo de execução.
  4. Memory Limit Exceeded (MLE) – o seu programa esgotou a restrição de memória utilizada.
  5. Run Time Error (RTE) – o seu programa foi interrompido devido a um erro em tempo de execução.

Tornando-se um programador competitivo

Dica 5: Aperfeiçoar sua habilidade em depurar códigos

Regras gerais para desenvolver bons testes de depuração de código:

- Seus testes devem incluir os testes de exemplo, geralmente presentes no enunciado. Utilize o comando `fc` em Windows ou `diff` em Linux para comparar se a saída do seu programa está igual à saída do enunciado.
- Para problemas com múltiplos casos de testes, inclua duas cópias consecutivas de um mesmo caso de teste. Se as saídas diferirem entre si, torna-se um indício de que as variáveis não foram inicializadas corretamente.
- Testar condições de contorno ($N = 0$, $N = 1$, $N < 0$, $N > 2^{32}$, etc.).
- Testar casos grandes, tanto com estruturas triviais (fáceis de avaliar) como estruturas geradas aleatoriamente (para avaliar robustez).

Tornando-se um programador competitivo

Dica 6: Praticar, praticar e praticar

- Assim como atletas de competição, um programador competitivo deve treinar regularmente para manter-se em "boa forma", ou seja, para que sua cultura em problemas de programação mantenha-se fresca na memória.
- Há diversos "juízes online" na internet para aprimorar suas habilidades em programação. Alguns deles são:
  1. http://uva.onlinejudge.org/
  2. http://livearchive.onlinejudge.org/
  3. http://www.spoj.com/
  4. http://br.spoj.com/
  5. http://www.urionlinejudge.com.br/
  6. http://acm.timus.ru/
  7. http://poj.org/
  8. http://acm.zju.edu.cn/onlinejudge/

Referências

1. S. Halim e F. Halim. Competitive Programming 2, Second Edition Lulu (www.lulu.com), 2011. (IMECC – 005.1 H139c)
2. S. S. Skiena, M. A. Revilla. Programming Challenges: The Programming Contest Training Manual, Springer, 2003.
3. T.H. Cormen, C.E. Leiserson, R.L.Rivest e C. Stein. Introduction to Algorithms. 2nd Edition, McGraw-Hill, 2001. (no. chamada IMECC – 005.133 Ar64j 3.ed.)
4. U. Manber. Introduction to Algorithms: A Creative Approach. Addison-Wesley. 1989. (no. chamada IMECC – 005.133 Ec53t 2.ed.)