Desafios de Programação

Prof. Eduardo Theodoro

Universidade Federal de Mato Grosso do Sul (UFMS)

Avaliação

A avaliação dos alunos será feita com base em duas provas e dois trabalhos práticos.

- P1 - Prova 1
- P2 - Prova 2
- T - Trabalho (Várias competições durante a disciplina - idealmente a cada duas semanas!)

Média Final (MF) calculada pela fórmula:

- MP = (P1 + P2)/2
- MF= (T*0.4 + MP*0.6)

Ementa

1. Estrutura de dados, ordenação, aritmética, combinatória, teoria dos números, divisão e conquista, backtracking, manipulação de cadeias de caracteres.
2. Algoritmos em Grafos.
3. Programação dinâmica.

Bibliografia

1. **HALIM, Steven; HALIM, Felix., Competitive Programming. Lulu. com, 2010.**
2. CORMEN, T. H. Algoritmos: teoria e prática. Elsevier, 2002.
3. SKIENA, S. S.; MIGUEL, A. R., Programming challenges: The programming contest training manual. Springer Science & Business Media, 2003.
4. Kleinberg, Jon; Tardos, Éva. Algorithm Design. Pearson Education India, 2006.

Onde praticar?

- http://uva.onlinejudge.org/
- https://www.urionlinejudge.com.br/
- http://www.programming-challenges.com/
- http://br.spoj.com/
- http://ahmed-aly.com/

Algumas dicas

- Digite rápido!
- Identifique rapidamente o tipo do problema.
- Faça a análise do algoritmo.
- Domine a linguagem de programação.
- Domine a arte de testar código.
- Pratique e pratique mais!
- Trabalho em equipe.

Algumas dicas

- Digite rápido!
- Identifique rapidamente o tipo do problema.
- Faça a análise do algoritmo.
- Domine a linguagem de programação.
- Domine a arte de testar código.
- Pratique e pratique mais!
- Trabalho em equipe.

Algumas dicas

- Digite rápido!
- Identifique rapidamente o tipo do problema.
- Faça a análise do algoritmo.
- Domine a linguagem de programação.
- Domine a arte de testar código.
- Pratique e pratique mais!
- Trabalho em equipe.

Algumas dicas

- Digite rápido!
- Identifique rapidamente o tipo do problema.
- Faça a análise do algoritmo.
- Domine a linguagem de programação.
- Domine a arte de testar código.
- Pratique e pratique mais!
- Trabalho em equipe.

Algumas dicas

- Digite rápido!
- Identifique rapidamente o tipo do problema.
- Faça a análise do algoritmo.
- Domine a linguagem de programação.
- Domine a arte de testar código.
- Pratique e pratique mais!
- Trabalho em equipe.

Algumas dicas

- Digite rápido!
- Identifique rapidamente o tipo do problema.
- Faça a análise do algoritmo.
- Domine a linguagem de programação.
- Domine a arte de testar código.
- Pratique e pratique mais!
- Trabalho em equipe.

Algumas dicas

- Digite rápido!
- Identifique rapidamente o tipo do problema.
- Faça a análise do algoritmo.
- Domine a linguagem de programação.
- Domine a arte de testar código.
- Pratique e pratique mais!
- Trabalho em equipe.

Algumas dicas

- Digite rápido!
- Identifique rapidamente o tipo do problema.
- Faça a análise do algoritmo.
- Domine a linguagem de programação.
- Domine a arte de testar código.
- Pratique e pratique mais!
- Trabalho em equipe.

Dica 1 - Digitar rápido

Praticar

`http://www.typingtest.com/`

Ideal é > 70 **wpm**!

Dica 2 - Identificar problemas

- Ad Hoc
- Busca
- Divisão e conquista
- Gulosos
- Programação Dinâmica
- Grafos
- Matemática
- Processamento de strings
- Geometria computacional

Dica 3 - Análise do tempo de execução do algoritmo!

- Computadores atualmente podem processar 10^6 operações por segundo!
- Se para determinado problema você desenvolveu um algoritmo $O(n^2)$, a entrada máxima de n é $100K$ $(K = 1000)$ e o tempo limite do problema é 3 segundos, o senso comum diz que o seu algoritmo estouraria o tempo limite!
- Se o seu algoritmo fosse $O(nlogn)$, o tempo de execução no pior caso seria na faixa de 1.7×10^6, o que passaria no tempo limite!

Dica 4 - Domine linguagens de programação

1. C++ (STL)
2. Java (BigInteger/BigDecimal, GregorianCalendar)

Dica 5 - Domine a arte de testar códigos

- Seus casos de teste devem incluir entradas que você já possui a resposta! Utilize **fc** no Windows ou **diff** no Linux para checar se a resposta do seu código corresponde a saída dada no problema.
- Para casos de teste com múltiplas entradas, inclua a mesma entrada duas vezes consecutivas. Isso ajuda a verificar se você esqueceu de inicializar alguma variável!
- Inclua casos de teste de 'borda', como N = 0 e N = máximo valor descrito no problema.
- Se possível, gera casos de testes aleatórios para verificar se seu programa termina dentro do tempo limite.

Dica 6 - Pratique e pratique mais!

- Competidores de programação são como atletas, eles precisam treinar regularmente para se manterem em forma!

Dica 7 - Trabalho em equipe

- Pratique escrever códigos em um papel (ajuda quando seu companheiro está utilizando o computador).
- Submeta e imprima seu código! Se seu código foi aceito, ignore a impressão. Caso contrário, use a impressão para debugar o código enquanto outra pessoa usa o computador.
- Se seu companheiro está escrevendo um algoritmo, ajude-o preparando casos de testes difíceis.

STL

Pode ser dividida em três grandes grupos:

1. Containers - classes utilizadas para armazenamento de dados, implementando uma coleção
2. Iteradores (iterators) - classes que permitem a varredura pelos elemento de uma coleção seguindo uma determinada regra.
3. Algoritmos - classes que implementam métodos para realização de algoritmos comuns de estruturas de dados sobre as coleções.

STL - Containers

Correspondem às coleções de elementos de um determinado tipo, na forma de gabaritos de classe. Os conteiners definidos pela STL são:

- vector - elementos organizados na forma de um array que pode crescer dinamicamente.
- list - elementos organizados na forma de uma lista duplamente encadeada.
- queue - implementação de um fila em C++.
- stack - implementação de uma pilha em C++.
- map - cada elemento é um par $< chave, elemento >$ sendo que a chave é usada para ordenação da coleção.
- set - coleção ordenada na qual os próprios elementos são utilizados como chaves para ordenação da coleção.

STL - Iteradores

Correspondem a objetos que podem ser utilizados para acessar os elementos de um container. Três categorias de iteradores são suportadas:

- iterator: permite percurso dos elementos do início para o fim da coleção
- reverse_iterator: permite percurso dos elementos na ordem inversa (do fim para o início) da coleção.
- random_access: acesso aleatório. Definido para os gabaritos vector.

STL - Algoritmos

Correspondem a gabaritos de funções que implementam algoritmos de estruturas de dados (definidos na biblioteca algorithm). As funções são divididas em 3 categorias:

- sequência: implementam operações de busca e alteração da sequência de elementos do container
- classificação: implementam classificação dos elementos do container
- numéricos: implementam funções numéricas comuns sobre os elementos do container.

STL - Algoritmos - Sequência

- **copy(start, end, dest)**: copia os elementos entre os iteradores start e end para dest.
- **fill(start, end, val)**: atribui val a todos os elementos entre os iteradores start e end
- **remove(start, end, val)**: remove todos os elementos de valor val entre os iteradores start e end.
- **replace(start, end, old_value, new_value)**: substitui os elementos iguais a old_value por new_value entre os iteradores start e end.
- **reverse(start, end)**: inverte a ordem dos elementos entre os iteradores start e end

STL - Algoritmos - Sequência

- **rotate(start, middle, end)**: rotaciona os elementos entre os iteradores start e end de tal maneira que o iterador middle fique posicionado onde antes estava o iterador start.
- **equal (start1, end1, start2)**: retorna true se os elementos entre os iteradores start1 e end1 são iguais aos da faixa de mesmo tamanho iniciado em start2.
- **find(start, end, val)**: retorna um iterador para a primeira ocorrência do elemento val entre os iteradores start e end.
- **search(start1, end1, start2, end2)**: procura o subconjunto dos elementos entre os iteradores start2 e end2 dentro do conjunto dos elementos entre start1 e end1.

STL - Algoritmos - Classificação

- **sort(start, end)**: classifica em ordem crescente os elementos entre os iteradores start e end, utilizando o algoritmo introsort, sem garantia de ordem entre os elementos de mesmo valor
- **stable_sort(start, end)**: semelhante a sort porém mantém a ordem original dos elementos que são iguais.
- **sort_heap(start, end)**: transforma o heap entre os iteradores start e end em um heap classificado (para criar um heap a partir de um container sequencial pode-se utilizar make_heap(start, end).

STL - Algoritmos - Numéricos

- **accumulate(start, end, val)**: acumula e retorna o valor de val com todos os valores entre os iteradores start e end.
- **count(start, end, val)**: conta quantos valores entre os iteradores start e end são iguais a val.