Guia: Listas IF672 - Algoritmos

Índice

1. Introdução às listas

Serão, a priori, 6 listas no total - compondo 30% da nota final. Os assuntos das listas serão:

1. Estruturas de dados lineares (pilhas, filas, array, [...])
2. Ordenação, busca e tabelas de dispersão (hashtables, busca binária, quicksort, mergesort)
3. Árvores (AVL, BST, [...])
4. Heaps, Conjuntos disjuntos (minheap, maxheap, DSU, [...])
5. Grafos, caminhos e árvores geradoras mínimas (BFS, DFS, Dijkstra, Prim, Floyd-Warshall, [...])
6. Programação dinâmica, backtracking (knapsack 0-1, branch and bound[...])

2. Monitoria

2.1 Monitorias presenciais

Horário das monitorias: a combinar no discord

É possível que façamos atividades valendo pontuação extra nas monitorias.

2.2 Discord da disciplina

O canal oficial de comunicação da monitoria é o Discord. Quando possível, enviar dúvidas diretamente no canal correspondente ao invés de esperar um monitor responder que está disponível.

Quando solicitar ajuda no código, deixe sua última submissão organizada e comentada, e se possível explique o problema.

Ao usar os canais públicos, não é permitido compartilhar código - entre em contato com os monitores no chat privado.

3. Apresentação do ambiente Iudex

3.1 Vereditos:

A: Aceito

- Tudo certo.

TLE: Tempo Limite Excedido

- A execução não foi finalizada dentro do limite definido pelo problema.

MLE: Limite de Memória Excedido

- A memória utilizada na execução ultrapassou os limites do problema.

RTE: Erro durante a execução

- Falha de segmentação (acesso a área de memória não alocada para o programa) é a causa mais comum. ver flags `fsanitize` na seção 5.

CE: Erro de compilação

- Se o código está rodando na sua máquina, deveria rodar no Iudex - testar antes de fazer a submissão.

WA: Resposta Errada

- A compilação e execução concluíram sem esbarrar nos limites de tempo e memória, porém a saída do seu programa está errada.

PE: Erro de apresentação

- Resposta correta, mas formatação errada: cuidado com linhas em branco

Note que ao esbarrar em uma condição de tempo limite excedido, erro na execução ou limite de memória excedido, o árbitro virtual interrompe a execução e retorna o veredito correspondente. Assim, é possível que, após resolver o problema imediato, a próxima submissão dê um erro diferente (como resposta errada, por exemplo).

Similarmente, o CE significa que a compilação não concluiu, portanto qualquer erro ainda é possível após corrigir o código de forma que ele compile corretamente.

3.2 Observações

- O peso de cada caso teste não é necessariamente igual. Tipicamente, os casos maiores valem mais.
- Apenas a última submissão é verificada, então enviem a melhor submissão por último.

4. Bibliotecas permitidas

4.1 C/C++:

- <stdlib.h>
- <stdio.h>
- (c++)

Note que o motivo pelo qual restringimos o uso de bibliotecas é para que vocês aprendam com a implementação dos algoritmos e estruturas de dados relevantes.

Nesse sentido, é possível que uma ou outra função de outras bibliotecas também seja permitida, desde que não abstraia uma parte fundamental do programa.

Pela mesma lógica, também é restrito o uso das funções sdlib::qsort() ou stdlib::bsearch() se a ideia do problema for implementar um algoritmo de ordenação ou busca.

4.2 Python

Muitas das funcionalidades presentes por padrão do Python possuem abstrações de alto nível, incompatíveis com a intenção didática das listas (entender implementações de relativo baixo nível dos algoritmos e estruturas de dados mais importantes), por isso é difícil listar exaustivamente as funções e estruturas que não podem ser utilizadas. Comunicaremos no discord as restrições de forma específica.

5. Comandos de terminal (C/C++)

5.1 Compilação

C:

```
gcc [nome do programa] // compila
./a.out // executa
```

a.out é o nome padrão do executável gerado na compilação. Para especificar um nome diferente, você pode usar a flag -o ('o' significa output):

```
gcc -o [nome do executável] [nome do programa]
./[nome do executável]
```

Exemplo:

```
gcc -o a lista2.c
./a
```

Obs: também é possível rodar C com o compilador clang e C++ com clang++

C++

muito semelhante a C, mas o compilador é, tipicamente, g++. Exemplo:

```
g++ -o a lista2.cpp
./a
```

5.2 Entrada e Saída

Para ler entrada a partir de um arquivo txt na execução:

```
./a.out < entrada.txt
```

Observação: o arquivo *entrada.txt* tem que estar no mesmo diretório que o executável.

escrevendo saída do programa em um arquivo:

```
./a.out > saida.txt
```

Para ler a partir de um arquivo e escrever em outro:

```
./a.out < entrada.txt > saida.txt
```

Comparar saída gerada com saída esperada

Fazer bom uso dos casos aparentes no iudex, faça isso em vez de comparar manualmente!!!!

```
diff arquivo1.txt arquivo2.txt
```

Outras extensões além de .txt também funcionam: .out, etc. O terminal então retorna as linhas onde há diferença entre os arquivos, ou retorna nada quando os arquivos são idênticos.

Observações

Aparentemente, é possível que o diff não funcione no Command Prompt do Windows, sendo necessário baixar pacotes adicionais: ver esse link

Outra opção é usar o diffchecker

5.3 Tempo de execução

Sintaxe:

```
time ./[executável]
```

Exemplo:

```
time ./a.out
```

Observações:

- tipicamente, os casos disponíveis para alunos no iudex são pequenos. Portanto, mesmo se os primeiros casos conseguem rodar em alguns milisegundos, não há garantia que seu código está eficiente e passará dentro do limite de tempo nos casos finais (tipicamente muito maiores).
- Esse comando de terminal mostra o tempo que o programa demorou na sua máquina, podendo rodar mais rápido ou mais devagar no iudex.

- Por isso, é mais importante pensar na complexidade assintótica do seu programa do que medir sua performance em casos específicos.

Como ele vai se comportar à medida que a entrada aumenta? O que acontece quando a entrada é o "pior" caso possível (maior número de operações)?

5.4 Flags

Especificar a versão do c++ para compilação:

```
g++ -o a a.cpp -stdc++17
```

(c++ 17 é a versão utilizada no iudex)

Versões disponíveis:

- -std=c++11 (ISO C++11)
- -std=c++14 (ISO C++14)
- -std=c++1z ou -std=c++17 (ISO C++17)
- -std=c++20 (C++20)

Especificar impressão de avisos na compilação:

Por default, muitos avisos são desativados na compilação. Para encontrar problemas é útil pedir ao compilador que imprima avisos:

- -Wall: imprime avisos
- -Wextra: imprime mais avisos
- -Werror: todo aviso é tratado como erro, compilação interrompe e mensagem de erro é imprimida

Exemplo:

```
g++ -o main main.cpp -Wall -Wextra
```

Mais detalhes

Para detectar erros de memória

Detecta vazamentos de memória, falhas de segmentação, etc em tempo de execução.

```
-fsanitze=address
```

Exemplo:

```
g++ main.cpp -o main -fsanitize=address
```

Mais detalhes

Comportamento indefinido:

Essa flag pode reportar comportamento indefinido em tempo de execução:

```
-fsanitize=undefined
```

Exemplo:

```
g++ main.cpp -o main -fsanitize=undefined
```

Algumas referências:

Opções de compilação Flags mais comuns

6. Erros comuns e dicas

1. Ler questão (de verdade)
2. Resolver ao longo do semestre mesmo que atrasado, evitar acumular
3. Entender solução antes de codar
4. Deixar margem de tempo antes da data de entrega para resolver bugs (mesmo quando o código está basicamente pronto)

6.1 Profiling

Em termos simples, Profiling é a análise de performance de código em tempo de execução. Entre outras coisas, ferramentas de profiling retornam o tempo gasto em cada função e o número de chamadas de funções. Isso é útil para encontrar ineficiências.

Profiling em C++

É preciso compilar o arquivo com uma flag específica, depois executar o programa e ler a saída do profiler:

```
g++ -pg -o my_program my_program.cpp // compila programa com flag -pg
./my_program // executa
gprof my_program gmon.out > analysis.txt // joga saída (gmon.out por padrão) em um arquivo txt
```

Profiling em Python

Ver explicação.

6.2 Otimizações

Observação

Acertar a complexidade assintótica do algoritmo é o mais importante. No entanto, algumas otimizações geralmente podem acelerar o tempo de execução (por um fator constante)

Entrada e saída

- scanf/printf é mais rápido do que cin/cout (geralmente)
- '/n' é mais rápido do que std::endl

Mais detalhes:

scanf/printf vs cin/cout endl vs /n

Observações:

- std::endl é útil para debugar: imprime durante a execução e não no final.
- Se usar cin/cout, adicionar as seguintes linhas no início da função main:

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

  Com essa otimização, diferenças de desempenho entre printf/scanf e cin/cout serão geralmente pequenas.

Algortimos iterativos vs recursivos

Embora muitas vezes algoritmos recursivos sejam mais intuitivos, é geralmente **preferível usar implementações iterativas** (que utilizam laços ao invés de chamadas recursivas) de algoritmos. Isso porque a cada chamada recursiva, o compilador tem que alocar memória adicional, o que pode ser custoso.

mais detalhes

Referências vs objetos como parâmetros de função

Os parâmetros recebidos pelas funções têm que ser alocados em algum lugar na memória. Por isso, é uma boa ideia evitar passar objetos grandes para dentro de funções, utilizando referências ou objetos acessíveis no escopo da função.

exemplo:

```
int busca(int array[], int pos) // mais devagar: o array inteiro é copiado a cada chamada da função
{
...
}

int array[1000000];
int busca(int pos) // mais rápido (acessa array global)
{
...
}

int busca(vetor& meu_vetor) // passa por referência (operador &): vetor não é copiado
{...}
```

6.3 Diagnosticando Runtime Errors (RTE)

Causas Comuns

- Acessos a regiões de memória proibidos (Ponteiro nulo, objetos fora de escopo, índice inválido de array)
- Variáveis não inicializadas (podem gerar comportamento inesperado em algum momento na execução)
- Divisão por zero

Flags de compilação

as flags `-fsanitize=address` e `fsanitize=undefined` são capazes de identificar grande parte dos erros em tempo de execução.

6.4 Gerando casos teste

É possível gerar casos automaticamente para testar o seu código utilizando um programa. Um exemplo de script para geração de casos está presente nesse repositório.

Para casos pequenos, também é possível pedir que o Chat GPT, ou equivalente, gere uma entrada (possivelmente incorreta) para o programa seguindo a especificação dada.

Para teste de algoritmos e estruturas de dados bem conhecidas, é possível buscar problemas relacionados no Leetcode e testar a sua implementação.

7. Referências úteis

Anotações completas - algoritmos: Anotações IF672 com Paguso (Pseudocódigo e análises de complexidade)

VisuAlgo: Visualizações de Algoritmos

Abdul Bari: Videoaulas

Geeks for Geeks: Implementações comentadas de algoritmos e estruturas

Competitive Programming Algorithms: Explicações e código de alto nível de abstração