Universidade Federal de Roraima
Departamento de Ciência da Computação
Análise de Algoritmos
DISCIPLINA: Análise de Algoritmos – DCC606

2a Lista

Prazo de Entrega: 18/06/2019

ALUNO(A): Larissa Santos Silva **NOTA:_____**

ATENÇÃO: Descrever as soluções com o máximo de detalhes possível, inclusive a forma como os testes foram feitos. Todos os artefatos (relatório, código fonte de programas, e outros) gerados para este trabalho devem ser adicionados em um repositório online no github, com o nome do repositório no seguinte formato: **nomeDoAluno_AA_Lista2_rr_2019**. Na resposta para as questões de implementação deve ser apresentado: o modo de compilar/executar o programa; a linha de comando para executar o programa; e um exemplo de entrada/saída do programa.

[QUESTÃO – 01]

Especifique cada problema e calcule o M.C. (melhor caso), P.C. (pior caso), C.M. (caso médio) e a ordem de complexidade para algoritmos (os melhores existentes e versão recursiva e não- recursiva) para problemas abaixo. Procure ainda, pelo L.I. (Limite Inferior) de tais problemas:

(A) N-ésimo número da sequência de Fibonacci

Pode ser encontrado com a função **recursiva com complexidade de 2^n**:

```
fib(n)
        if(n >=1)
                return 1
        else
                fib(n-1) + fib(n-2)
```

Cuja análise assintótica gera a equação T(n) = T(n-1) + T(n-2) + c (uma constante).

O limite inferior pode ser calculado aproximando o valor de F(n-1) a F(n-2).

T(n) = T(n-1) + T(n-2) + c
= 2T(n-2) + c
= 2*(2T(n-4) + c) + c
= 4T(n-4) + 3c
= 8T(n-8) + 7c
= 2^k * T(n - 2k) + (2^k - 1)*c
n - 2k = 0

k = n/2
T(n) = 2^(n/2) * T(0) + (2^(n/2) - 1)*c
= 2^(n/2) * (1 + c) - c

T(n) ~ 2^(n/2)
Complexidade O(2^(n/2))

Na sua forma iterativa:

```
fibo(int n)
 if(n <= 1){
          return n;
 int fibo = 1;
 int fiboPrev = 1;
 for(int i = 2; i < n; ++i)
          int temp = fibo;
          fibo += fiboPrev;
          fiboPrev = temp;
 return fibo;
```

Somando temos 3*n + 3, o que gera uma complexidade de O(n)

(B) Geração de todas as permutações de um número

O escolhido foi o Heap, pois ele tenta reduzir o número de computações ao trocar somente dois elementos de cada vez e garante chegar a todas as permutações. Ele começa definindo um contador i = 0 e vai iterar sobre até gerar (n - 1)! permutações que terminam com cada número da sequência, se o valor de n for ímpar trocamos o primeiro termo com o último e se for par trocamos o i-ésimo termo com o último.

//disponível em: https://en.wikipedia.org/wiki/Heap%27s_algorithm

```
procedure generate(n : integer, A : array of any):
   if n = 1 then
       output(A)
   else
      for i := 0; i < n - 1; i += 1 do
         generate(n - 1, A)
         if n is even then
            swap(A[i], A[n-1])
         else
            swap(A[0], A[n-1])
         end if
      end for
   end if
```

if n = 1 { T(n) = 1 }
if n > 1 { T(n) = n*T(n-1) + 2 } Cuja complexidade é O(n!)

Versão iterativa:

```
procedure generate(n : integer, A : array of any):
   c : array of int


   for i := 0; i < n; i += 1 do
      c[i] := 0
   end for
   output(A)
   i := 0;
   while i < n do
      if  c[i] < i then
         if i is even then
            swap(A[0], A[i])
         else
            swap(A[c[i]], A[i])
         end if
         output(A)


         c[i] += 1
         i := 0
      else
         c[i] := 0
         i += 1
      end if
   end while
```

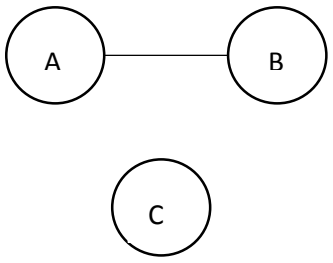
Complexidade de n*1 + n * 4 (levando em consideração if == verdadeiros)
O(n)

[QUESTÃO – 02]
Defina e dê exemplos:

(A) Grafos.

Um grafo G, é formado por dois conjuntos de elementos, onde o primeiro conjunto A são os vértices e o segundo conjunto B são as arestas, sendo definido como G = (A, B). Cada aresta liga dois vértices, e cada vértice pode estar ligado a nenhum ou a vários outros vértices.

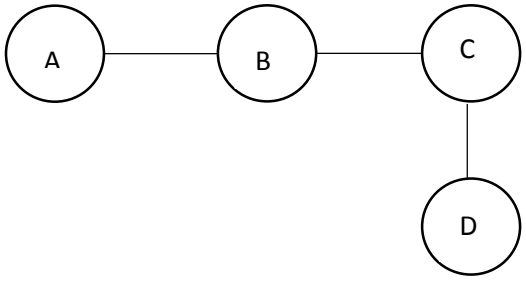
Ex.



(B) Grafo conexo, acíclico e direcionado.

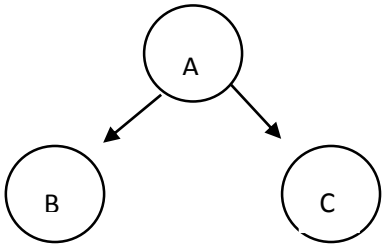
O grafo e conexo quando existe pelo menos um caminho entre qualquer par de vértices.

Ex.



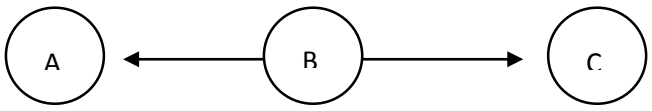
O grafo e acíclico, quando os caminhos entre os vértices não contenha vértices repetidos.

Ex.



O grafo e direcionado, quando as arestas possuem sentido, ou seja, não é permitido percorrer pelas arestas no sentido contrário.

Ex.



(C) Adjacência x Vizinhança em grafos.

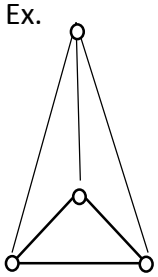
Refere-se aos vértices, dois vértices são adjacentes se existe uma aresta ligando os dois.

Ex.

(D) Grafo planar.

Um grafo $G(V,A)$ é dito ser planar quando existe alguma forma de se dispor seus vértices em um plano de tal modo que nenhum par de arestas se cruze.

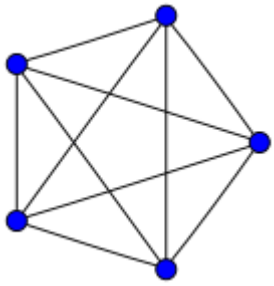
Ex.



(E) Grafo completo, clique e grafo bipartido.

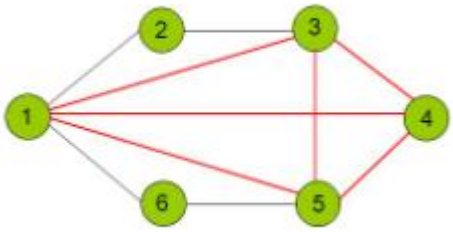
Grafo completo quando há uma aresta entre cada par de seus vértices. Estes grafos são designados por Kn, onde n é a ordem do grafo.

Ex.



Grafo clique é um subgrafo completo

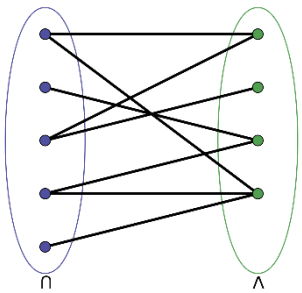
Ex.



Grafo bipartido

Um grafo Kn possui o número máximo possível de arestas para um dados n. Ele é, também regular-(n-1) pois todos os seus vértices tem grau n-1

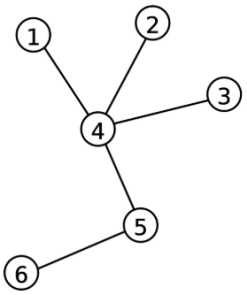
Ex.



(F) Grafos simples x multigrafo x digrafo.

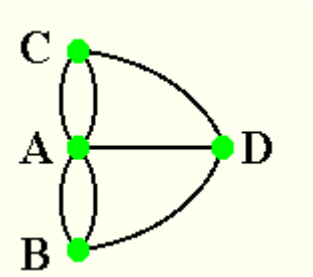
Gráfico simples não passa duas vezes pela mesma aresta (arco).

Ex.



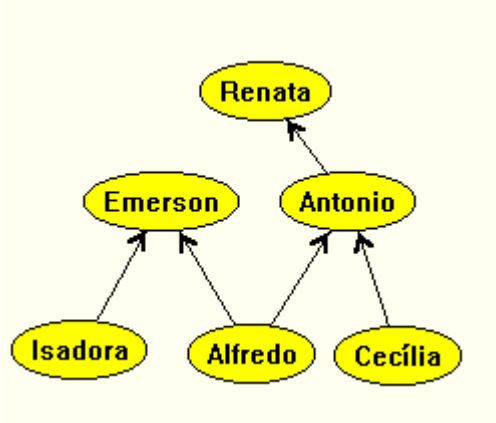
Multigrafo quando existem múltiplas arestas entre pares de vértices de G. No grafo G8, por exemplo, há duas arestas entre os vértices A e C e entre os vértices A e B, caracterizando-o como um multigrafo.

Ex.



Dígrafo pode mais de uma aresta para cada par de vértices e essas arestas possuem direção.

Ex.

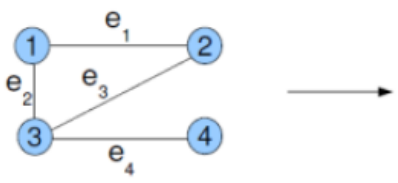


[QUESTÃO – 03]

Defina e apresente exemplos de matriz de incidência, matriz de adjacência e lista de adjacência. Adicionalmente, descreva o impacto (vantagens e desvantagens) da utilização de matriz de adjacência e lista de adjacência.

Matriz de incidência

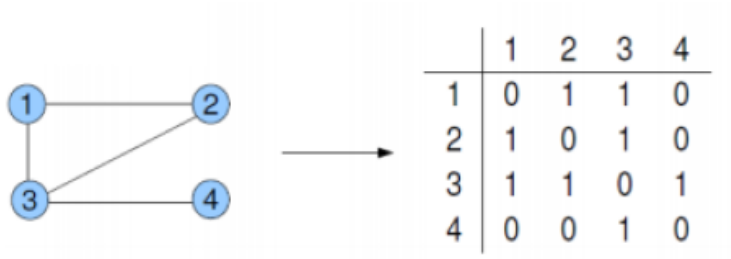
A matriz de incidência utiliza uma matriz m x n, onde n é o número de vértices e m é o número de arestas. Para essas matrizes, os vértices são as linhas e as arestas são as colunas e cada elemento da matriz indica se aresta incide sobre o vértice. Consegue proporcionar um acesso constante, porém a utilização de matrizes demanda muito espaço.



	e_1	e_2	e_3	e_4
1	1	1	0	0
2	1	0	1	0
3	0	1	1	1
4	0	0	0	1

Matriz de adjacência:

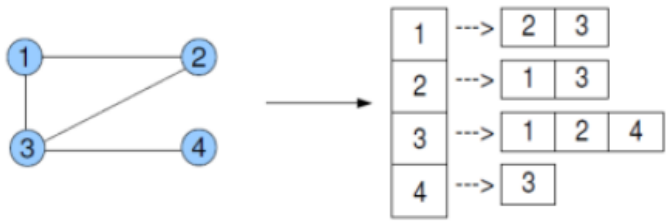
A matriz de adjacência de um grafo com |V| vértices é uma matriz |V|x|V| de 0s e 1s, na qual a entrada na linha i e coluna j é 1 se e somente se a aresta (i,j)(i,j) estiver no grafo. Uma vantagem da utilização de matriz de adjacência está no tempo constante de acesso (O(n)), e uma das desvantagens é a grande necessidade de espaço (Θ(V^2)).



	1	2	3	4
1	0	1	1	0
2	1	0	1	0
3	1	1	0	1
4	0	0	1	0

Lista de adjacência:

A representação de um grafo com listas de adjacência combina as matrizes de adjacência com as listas de arestas. Para cada vértice do grafo, existe uma lista encadeada que armazena os vértices adjacentes. Uma vantagem é a economia de espaço, porém existe um aumento no tempo de acesso.



[QUESTÃO – 04]

Defina, explicando as principais características e exemplifique:

(A) Enumeração explícita x implícita.

O método de enumeração explicita, faz uma enumeração de todas as possíveis soluções, já em enumeração implícita faz uma enumeração “inteligente”, com as melhores soluções para o problema.

(B) Programação Dinâmica.

É um esquema de enumeração de soluções, em utiliza de uma técnica de decomposição minimizar o montante de computação, assim evita que um mesmo subproblema seja resolvido diversas vezes. Resolve o subproblema uma vez e reutiliza a solução toda vez que o mesmo problema aparece.

(C) Algoritmo Guloso.

Sempre fazer a melhor escolha local para ter uma melhor escolha global, geralmente atingindo soluções bem próximas da ótima, porém em alguns casos não encontra soluções tão ótimas. Um exemplo de utilização de algoritmos gulosos é encontrar uma solução do problema da mochila fracionária.

(D) Backtracking.

É um tipo de algoritmo que representa um refinamento da busca por força bruta, em que múltiplas soluções podem ser eliminadas sem serem explicitamente examinadas.

[QUESTÃO – 05]

Implemente uma solução para multiplicação de matrizes utilizando programação dinâmica, visando determinar uma ordem em que as matrizes sejam multiplicadas, de modo a minimizar o número de multiplicações envolvidas.

```
//Algoritmo de Strassem para multiplicação de Matrizes

//O algoritmo foi testado na plataforma https://www.jdoodle.com/c-online-compiler na linguagem C.

//Referenca em https://www.sanfoundry.com/c-program-implement-strassens-algorithm/

#include<stdio.h>

#define TAM 2

int main(){

 int A[TAM][TAM], B[TAM][TAM], C[TAM][TAM], i, j;

 int M1, M2, M3, M4 , M5, M6, M7;


 printf("Entre com os 4 valores da primeira matriz: ");

 for(i = 0;i < TAM; i++)

    for(j = 0;j < TAM; j++)

        scanf("%d", &A[i][j]);


 printf("Entre Com os 4 valores da segunda matriz: ");

 for(i = 0; i < TAM; i++)

    for(j = 0;j < TAM; j++)

        scanf("%d", &B[i][j]);


 printf("\nPrimeira matriz: \n");
```

```
for(i = 0; i < TAM; i++){
   printf("\n");
   for(j = 0; j < TAM; j++)
       printf("%d\t", A[i][j]);
}

printf("\nSegunda matriz\n");
for(i = 0;i < TAM; i++){
   printf("\n");
   for(j = 0;j < TAM; j++)
       printf("%d\t", B[i][j]);
}

M1= (A[0][0] + A[1][1]) * (B[0][0] + B[1][1]);
M2= (A[1][0] + A[1][1]) * B[0][0];
M3= A[0][0] * (B[0][1] - B[1][1]);
M4= A[1][1] * (B[1][0] - B[0][0]);
M5= (A[0][0] + A[0][1]) * B[1][1];
M6= (A[1][0] - A[0][0]) * (B[0][0]+B[0][1]);
M7= (A[0][1] - A[1][1]) * (B[1][0]+B[1][1]);

C[0][0] = M1 + M4- M5 + M7;
C[0][1] = M3 + M5;
C[1][0] = M2 + M4;
C[1][1] = M1 - M2 + M3 + M6;

 printf("\nResultado da multiplicacao: \n");
 for(i = 0; i < TAM ; i++){
   printf("\n");
```

```
    for(j = 0;j < TAM; j++)
        printf("%d\t", C[i][j]);
  }
```

[QUESTÃO – 06]

Defina e exemplifique:

(A) Problema SAT x Teoria da NP-Completude.

Problemas SAT ou Problemas de satisfabilidade booleana, é o primeiro problema classificado com a complexidade NP-Completo, estar nessa classe de complexidade diz duas coisas sempre, primeiro, que ele está na classe NP e segundo, todos os outros problemas na classe NP poderiam ser reduzidos à esse problema em tempo polinomial, ao ponto que se for possível resolvê-lo, seria possível resolver todos os outros problemas nessa classe em tempo polinomialmente definido.

O problema de Satisfabilidade diz que, dado uma sequência de elementos booleanos (ex.: A ∧ ¬ B) procura-se saber se existem valores possíveis para substituir as variáveis de modo a gerar uma solução verdadeira (TRUE), nesse caso ela é satisfatória. Caso não exista tal combinação, ela retorna falso (FALSE) e é insatisfatória. Esse problema poderia ser pensando na forma de um grafo se adaptássemos as sentenças para uma forma que exprima implicância, exemplo (a v b) pode escrito como (~a -> b v ~b -> a). Daqui podemos escrever toda uma sentença em forma de grafo e fazer uma pesquisa nesse grafo para achar uma inconsistência, como ~b -> b, que classificaria o problema como insatisfatório e, caso contrário, satisfatório.

(B) Classes P, NP, NP-Difícil e NP-Completo.

A classe P de complexidade abrange problemas cujo as soluções podem ser encontradas em tempo polinomial determinístico.

Exemplo, o problema de determinar se um número é primo.

NP (polinomial não determinista) é a classe de problemas cuja solução pode ser verificada a partir de uma entrada e retornar um “sim” ou “não” em um tempo limitado por um polinômio, assim chamados problemas polinomialmente verificáveis.

Exemplo, o problema do isomorfismo de sub grafos: dados dois grafos A e B como input, quer-se determinar se o B é sub grafo de A.

Também dentro de NP existe uma classe chamada NP-Completo cujos elementos atendem a duas condições:

- estar dentro da classe NP, como já dito.
- todo problema em NP é redutível para esse problema em tempo polinomial. De modo que se conseguíssemos achar uma solução em tempo polinomial para esse problema, poderíamos resolver todos os outros problemas em NP em tempo polinomial.

O próprio problema de satisfabilidade, assim como o problema de Coloração de grafos que se propõe a descobrir como colorir os vértices de um grafo baseado em alguma restrição (geralmente que duas cores não podem estar justapostas).

NP-Difícil é a classe dos problemas “tão difíceis quanto NP-completo” por que mesmo não estando na classe NP e mesmo sem nem precisar ser problemas de decisão, estes problemas são reduzíveis a problemas NP.

Exemplo, o problema da parada que é um problema de decisão que pergunta se dado um programa ele vai terminar ou rodar para sempre.

[QUESTÃO – 7]
Descreva a redução (prove a NP-Completude) do problema do SAT ao Clique. Apresente o pseudo-código do algoritmo NP e mostre graficamente as instâncias e soluções, no processo de redução.

Para provar a NP-completude de um problema é preciso provar que esse problema está em NP, ou seja, gastaria tempo polinomial para checar uma possível solução, e se todo problema em NP é redutível para tal problema em tempo polinomial. No caso ao reduzir SAT em um problema de clique, estaríamos provando que achar uma solução Y para SAT estaria na mesma ordem de dificuldade do que achar uma solução X para o Clique.

Suponhamos que temos a fórmula booleana a qual queremos provar sua satisfabilidade.

$$(a \vee \neg b \vee c) \wedge (\neg a \vee b \vee c) \wedge (a \vee \neg b \vee \neg c)$$

Poderíamos transformar um grafo bidirecionado que atendesse a restrição de não possuir arestas que causem contradição, por exemplo “a” ligando com ¬a.