Trabalho 1 – Estruturas Discretas
Lucas Hardman – 1113567
Stephanie Fay – 1121598

Exercício 1

(a)

Algoritmo em pseudocódigo:

```
função quociente (x, y, k){
        se (k igual a 1)
                então retorna 1
        retorna x^(k-1)+ y*q[k-1]
}
```

(b)

Algoritmo em Python:

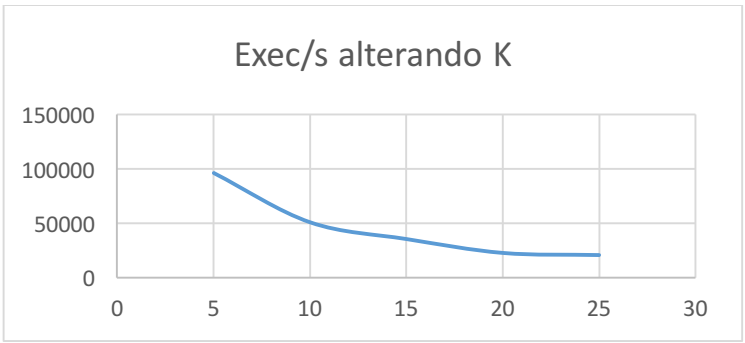
```
def quociente (x, y, k):
        if k == 1:
                return 1 # para quando q[k - 1] = 1
        return x**(k - 1) + (y * quociente (x, y, k - 1)) # q[k] = x^(k - 1) + y*q[k - 1]
```

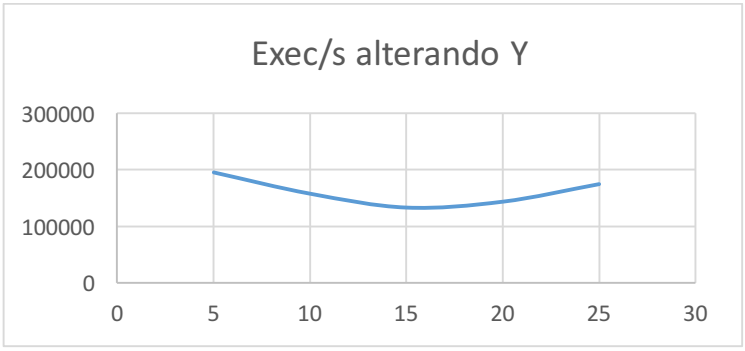
Testes:

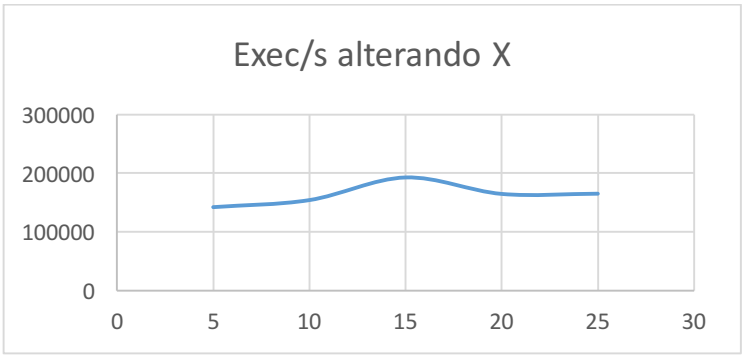
X	Y	K	Q[K]	Tempo	Execuções	Execuções por segundo
1	2	5	31	5,00028610	481317	96257
1	2	10	1023	5,00028610	255345	51066
1	2	15	32767	5,00028586	178564	35710
1	2	20	1048575	5,00028586	114964	22991
1	2	25	33554431	5,00028586	105155	21029
1	5	2	6	5,00028610	978300	195648
1	10	2	11	5,00028610	787903	157571
1	15	2	16	5,00028586	663620	132716
1	20	2	21	5,00028586	716388	143269
1	25	2	26	5,00028610	873504	174690
5	1	2	6	5,00028586	711613	142314
10	1	2	11	5,00028586	771478	154286
15	1	2	16	5,00028586	963514	192691
20	1	2	21	5,00028610	823931	164776
25	1	2	26	5,00028586	825827	165155

Conclusão:

Abaixo estão três gráficos para ilustrar a diferença da mudança dos valores de X, Y e K em relação a quantidade de execuções por segundo. Eixo Y = execuções por segundo e eixo X = X, Y ou K.







Podemos observar no gráfico “Exec/s alterando K” que a quantidade de execuções por segundo diminui bastante quando aumentamos o valor de K, saindo da faixa de 100 mil para 20 mil. O valor de K influencia diretamente (de 1 para 1) na quantidade de vezes que a função é chamada, uma vez que a recursão é chamada sempre com k-1 até que k == 1. Por exemplo, se K = 50 a função vai ser chamada 50 vezes.

Já ao aumentar os valores de X ou Y, observamos pouca alteração na quantidade de execuções por segundo. Alterar os valores de X e Y não influencia na quantidade de recursões da função, a pouca alteração é causada por fatores externos ao programa, como por exemplo outros processos sendo executados em paralelo. Nos gráficos, as quantidades de execuções por segundo estão sempre na faixa de 125 mil até 200 mil.

Exercício 2:

(a)

Temos um grafo G = (V, E) que é conexo, sem direção e sabemos que d(v) define o grau do vértice v.

Esse grafo G possui n>=3 vértices e cada par de vértices não adjacentes v e w satisfaz d(v) + d(w)>=n, então existe um ciclo Hamiltoniano em G.

Teorema do caso base:

O caso base é o grafo completo. Cada grafo completo com pelo menos três vértices contém um ciclo Hamiltoniano e é fácil de achar um (ponha todos os vértices em uma ordem arbitrária e conecte eles em um ciclo).

Teorema do passo indutivo:

Hipótese Indutiva: Sabemos achar um ciclo Hamiltoniano em grafos satisfazendo as condições com m arestas.

Temos que mostrar como achamos um ciclo Hamiltoniano em um grafo com m-1 arestas que satisfaça as condições do problema. Seja G=(V,E) o tal grafo. Pegue qualquer par de vértices **não-adjacentes** v e w em G, e considere o grafo G', que é o mesmo que G exceto que v e w estão conectados. Pela hipótese indutiva sabemos como achar um ciclo Hamiltoniano em G'. Seja x1, x2,..., xN, x1 o ciclo em G'. Se a aresta (v,w) não está incluída no ciclo, **então o mesmo ciclo está contido em G**. Senão, sem perder a a parte genérica, assumimos que v = x1 e w = xN. Pelas condições dadas por G, d(v) + d(w) >= n.

Considere todas as arestas de G saindo de v e w, existem ao menos n delas (pelas condições do problema). Mas G contém n-2 outros vértices. Portanto tem dois vértices xi e x(i+1), que são vizinhos no ciclo, como v estar conectado a x(i+1) e w estar conectado a x(i). Usando as arestas (v, x(i+1)) e (w, xi) podemos achar o novo ciclo Hamiltoniano que não usa a aresta (v, w). É o ciclo v(=x1), x(i+1), x(i+2), ..., w(=xN), xi, x(i-1), ..., v

(b)

Implementação: A implementação direta dessa prova começa com um grafo completo e substitui uma aresta de cada vez. Suponha um input de um grafo G, ache o maior caminho (DFS) e adicione as arestas (que não são de G) necessárias para completar esse caminho em um ciclo Hamiltoniano. Agora temos um grafo maior G', que tem um ciclo Hamiltoniano. Normalmente poucas arestas serão adicionadas, mas em um pior caso até n-1 arestas podem ser adicionadas. Podemos aplicar essa prova iterativamente, começando com G' até um caminho Hamiltoniano é feito em G. O número total de passos para substituir uma aresta é O(n). E há O(n) arestas para substituir. Logo o algoritmo tem O(n^2).

Algoritmo em Python:

```
def find_cycle(graph):
# função destinada a achar os ciclos Hamiltonianos
   cycles=[]
# variável destinada para guardar os ciclos
   for startnode in graph:
# pega o primeiro vértice, que será o ponto de partida
      for endnode in graph:
# e o último vértice para verificar o final do caminho (ciclo)
         newpaths = find_all_paths(graph, startnode, endnode)
# chama a função de achar todos os caminhos de um certo grafo, passando como parametro o começo e o final deste
         for path in newpaths:
# então para cada um dos caminhos achados
            if (len(path)==len(graph)):
# verificamos se o grau deste é igual ao tamanho total do grafo (caracterizando um ciclo)
               if path[0] in graph[path[len(graph)-1]]:
# se o caminho inicial está dentro do grafo
                  path.append(path[0])
# esse caminho é adicionado à estrutura de caminhos
                  cycles.append(path)
# e o ciclo Hamiltoniano reconhecido
   return cycles
# ao final de toda a iteração é retornada uma estrutura que possui todos os ciclos Hamiltonianos possíveis
```

(c)

Apenas o arquivo (hamilton_10_1.txt) e menores (criamos um hamilton_5_1.txt) foram lidos . Os demais arquivos não terminaram de executar (não foi possível saber se entram em loop infinito ou se iriam demorar um tempo muito maior do que o esperado para executar). Tivemos problemas para implementar uma função para ler os arquivos .txt e converter para um formato de grafo que nosso algoritmo find_cycle fosse capaz de interpretar de forma eficiente e rápida.

Exercício 3:

(a)

Temos que:
o numero de rodadas é r = 2^k – 1,
o numero de jogos é j = 2^(k – 1),
o numero de equipes é n = 2^k.

Teorema do caso base:

-> k = 1
r = 2^1 – 1 = 1
j = 2^(1 – 1) = 1
n = 2^1 = 2

Ou seja, o caso base é duas equipes jogando uma rodada de um jogo. Como são duas equipes e apenas um jogo, logo uma equipe está competindo contra a outra. Ou seja, nesta rodada (que é apenas um jogo) as equipes estão enfrentando uma equipe diferente.

Teorema do passo indutivo:

Hipótese indutiva: assumimos que o teorema 3 é válido para um k qualquer.

Queremos mostrar que o teorema também é válido para k + 1. Ou seja,
n = 2ˆ(k+1),
r = 2ˆ(k+1)-1.
j = 2ˆ(k+1-1) = 2ˆk.

Dividimos as equipes em dois grupos, A e B, de 2ˆk equipes cada.

Na primeira etapa, tratamos cada grupo como torneios diferentes e **ao mesmo tempo.** Pela hipótese indutiva temos dois torneios, de 2ˆk - 1 rodadas cada e 2ˆ(k - 1) jogos. Como as rodadas de cada torneio são simultâneas, temos ao todo 2ˆk - 1 rodadas com 2*(2ˆ(k - 1)) = 2ˆk jogos cada.

Ao fim da primeira etapa, todas equipes de A e B jogaram contra todas as outras equipes de seus respectivos grupos. Resta que cada equipe de A jogue contra cada equipe de B.

Para a segunda etapa, vamos considerar o grupo A como uma fila e o grupo B como uma fila circular.

A cada rodada da segunda etapa, a i-ésima equipe de A jogará contra a i-ésima equipe de B, para um total de 2ˆk jogos por rodada (afinal, A e B, cada um, tem 2ˆk equipes).

Ao final de cada rodada, **andamos uma posição na fila circular de B**, e repetimos o processo até que voltemos à posição inicial de B. Nesse ponto, todas as equipes de A terão jogado contra todas as equipes de B. Observe que serão 2ˆk rodadas nessa segunda etapa (pois B tem 2ˆk posições/equipes).

Assim, **a primeira e segunda etapas juntas terão (2ˆk - 1) + 2ˆk = 2ˆ(k + 1) – 1 = r rodadas, com 2ˆk = j jogos cada,** como desejado.

(b)

Algoritmo em pseudocódigo:

```
função geraTorneio(k, times){
        se times == nulo {
                times = [1, ... , 2*k]
        }
        se k == 1{ //Caso base
                jogo <- par(times[0], times[1])
                retorna jogo
        }
        grupo1 <- [1, ... , 2^(k-1)] //grupo 1 recebe a primeira metade de times
        grupo2 <- [2^(k-1), ... , 2^k] //grupo 2 recebe a segunda metade times

        t1 <- geraTorneio(k-1, grupo1)
        t2 <- geraTorneio(k-1, grupo2)
        torneio <- par(t1, t2) // Aqui é a primeira fase onde o primeiro time de A joga contra o primeiro time de B, o segundo de A contra o segundo de B e assim em diante.

        //Aqui começa a segunda fase, temos uma rotação dos times de B, de modo que o primeiro time de A enfrentará o ultimo time de B: [a1, b2^k], [a2, b1], ..., [a2^k, b2^k-1]
        para z de 0 ate tamanho(grupo1) – 1{
                round <- [[vazio]]
                para i de 0 ate tamanho(grupo1) – 1{
                        index1 = i
                        index2 = (i + z) % tamanho (grupo2)
                        jogo <- par(grupo1[index1], grupo2[index2])
                        round <- jogo
                }
                torneio <- round
        }
}
```

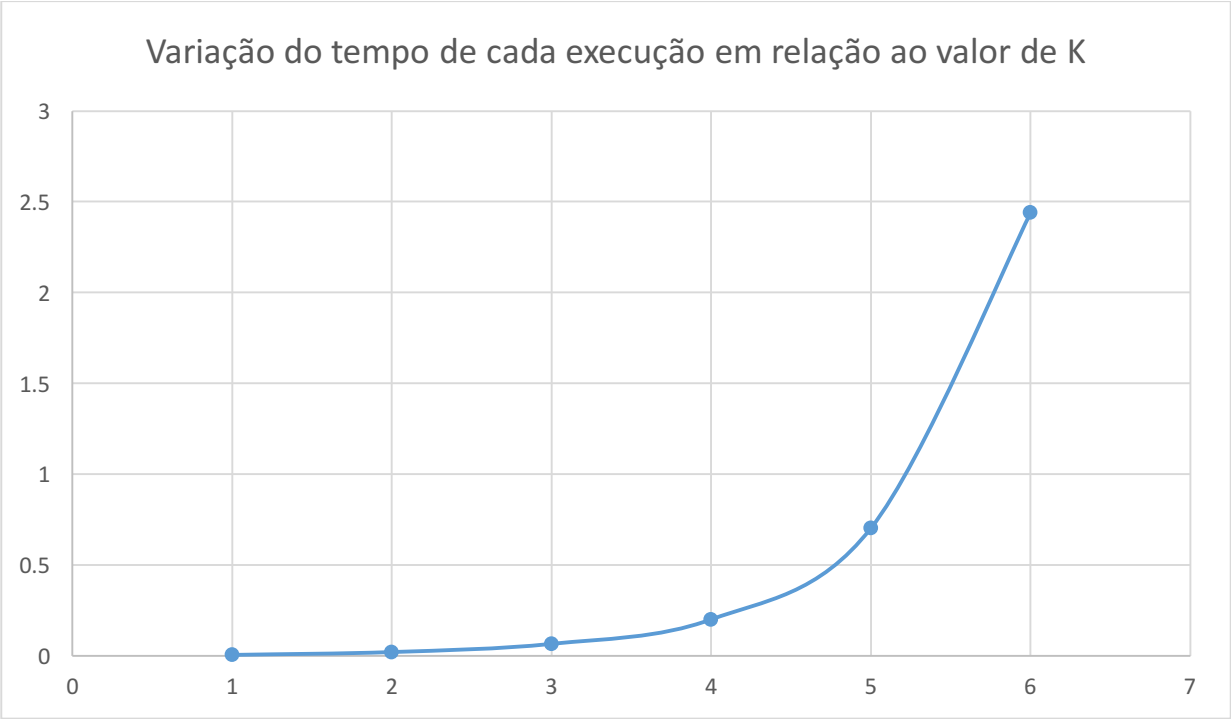
Algoritmo em Python:

```
def geraTorneio(k, start = 1): # start = o nomero de jogos
    if k == 1:
        return [[(start, start+1)]] # Retorna o par de times para a rodada 1 (caso base)
    t1 = geraTorneio(k-1, start) #t1 = tabela com os jogos (par de times)
    t2 = geraTorneio(k-1, 2**(k-1)+start) #t2 = tabela com os jogos na ordem inversa ao t1
    t = [] #tabela com todos os jogos

    #Divide o os times em dois grupos A e B
    for i in range(len(t1)):
        t.append(t1[i] + t2[i]) #Distribui t1 e t2 em uma tabela t
    times1 = range(start, 2**(k-1)+start) #times1 = "Grupo A" de times
    times2 = range(2**(k-1)+start, 2**(k)+start) #times2 = "Grupo B" de times

    #Inicia os jogos:
    #Time 1 do grupo A joga contra Time 1 do grupo B
    #Assim em diante ate acabarem os times
    #Quando acabam os times, temos uma rotação dos times de B, de modo que o primeiro time de A enfrentará o ultimo time de B: [a1, b2^k], [a2, b1], ..., [a2^k, b2^k-1]
    #Quando o time 1 do grupo A joga contra todos os times do grupo B, o mesmo se repete para todos os proximos times do grupo A
    #No codigo a seguir cuidamos do ciclo responsavel pela configuracao dos jogos discrita
    for z in range(len(times1)):
        round = []
        for i in range(len(times1)): #Anda com o grupo A
            index1 = i #Serve para andar com o grupo A (i anda normal)
            index2 = (i + z) % len(times2) #Serve para andar o grupo B (faz o grupo B andar em ciclo)
            game = (times1[index1], times2[index2]) #Passa o resultado para uma tabela
            round.append(game)
        t.append(round)
    return t #Retorna a tabela com todos os jogos
```

(c)

K = 12 foi o maior valor que o algoritmo foi capaz de gerar as rodadas. A seguir temos uma tabela e um gráfico representando os testes realizados.

K	Tempo (s)	Execuções (exec)	Tempo de cada execução (exec/ms)
1	5.008	1273802	0.003931
2	5.004	222260	0.022514
3	5.008	77236	0.064835
4	5.008	25080	0.199665
5	5.008	7124	0.702921
6	5.008	2051	2.441545
7	5.008	480	10.432519
8	5.008	125	40.060871
9	5.070	32	158.437781
10	5.242	5	1048.321819
11	5.117	2	2558.404446
12	9.766	1	9765.617132
13	MemoryError	MemoryError	MemoryError



Observamos que para cada vez incrementamos K o tempo de execução do algoritmo aumenta muito. Isto acontece pois n = 2^k, ou seja, a quantidade de equipes depende exponencialmente de K. Conforme a quantidade de equipes cresce, o número de jogos também cresce (pois todas as equipes devem se enfrentar). Lembramos que dividimos o torneio em 2, A e B, e isto é feito através de chamar o

algoritmo novamente (2 vezes, uma para A e outra para B). Desta forma, temos que aumentar o valor de K faz com que haja recursão duas vezes.

O algoritmo só foi capaz de gerar as rodadas para K < 13. Para os valores K >= 13 houve erro de memória, provavelmente porque o algoritmo não foi capaz de armazenar as informações necessárias para gerar tantas rodadas.

A seguir temos um exemplo da execução do código para K = 2.

```
C:\Users\180050202\Desktop\INF1631_Estruturas_Discretas\Trabalho1>py teorema3.py

       Digite um valor k > 1 fazer um teste:
k: 2
Faltam 5 segundos...
Faltam 4 segundos...
Faltam 3 segundos...
Faltam 2 segundos...
Faltam 1 segundos...

Execucoes: 222260
Tempo: 5.004s
Tempo por Execucao: 0.022514ms

Digite 0 caso queira ver o resultado: 0
              Round #01
              Game #01:              1 vs 2
              Game #02:              3 vs 4
              Round #02
              Game #01:              1 vs 3
              Game #02:              2 vs 4
              Round #03
              Game #01:              1 vs 4
              Game #02:              2 vs 3
```