Estruturas Discretas - Trabalho 1

Guilherme Berger Gabriel Siqueira Felipe Luiz

23 de Abril de 2014

Observação: o código fonte completo de todos os algoritmos em Python também foi enviado em anexo, para melhor leitura.

1 Teorema 1

Dado o teorema 1 e sua prova indutiva, deseja-se um algoritmo que, dados x, y e k, determine o quociente descrito.

Teorema 1 (k): $x^k - y^k$ é divisível por $x - y$ para quaisquer x e y inteiros e todos os valores de k inteiros e maiorez que zero.

O quociente supracitado é, então, $\frac{x^k - y^k}{x - y}$ nas condições acima mencionadas.

O algoritmo pode ser derivado diretamente da prova indutiva já fornecida.

Do caso base, sabemos que para $k = 1$, o quociente é igual a 1.

Também diretamente da argumentação fornecida, tem-se que:

$$x^{k+1} - y^{k+1} = q_{k+1} * (x - y) \tag{1}$$

$$q_{k+1} = (x^k + y * q_k) \tag{2}$$

Algoritmo genérico:

```
função quociente(x, y, k)
    se k == 1
        retorna 1
    retorna x^(k-1) + y * quociente(x, y, k - 1)
```

Implementação em Python:

```
def quo(x, y, k):
    if k == 1:
        return 1
    return x**(k-1) + y*quo(x, y, k-1)
```

Testes: A tabela abaixo ilustra os resultados dos testes:

x	y	k	Execuções	Tempo Total	Tempo/Execução
2	1	1	26100000	5.015 s	0.000192 ms
2	1	2	6500000	5.054 s	0.000778 ms
2	1	3	4000000	5.084 s	0.001271 ms
2	1	3	3900000	5.014 s	0.001286 ms
2	1	4	2800000	5.040 s	0.001800 ms
2	1	5	2200000	5.068 s	0.002303 ms
2	1	7	1500000	5.099 s	0.003399 ms
2	1	10	1000000	5.034 s	0.005034 ms
2	1	15	700000	5.624 s	0.008034 ms
2	1	50	200000	6.170 s	0.030851 ms
3	2	1	25700000	5.013 s	0.000195 ms
4	3	1	22900000	5.011 s	0.000219 ms
10	9	1	23500000	5.019 s	0.000214 ms
100	99	1	25200000	5.019 s	0.000199 ms
3	2	5	2000000	5.194 s	0.002597 ms

Conclusão: Observando os resultados dos testes executados e através da propria análise do algoritmo utilizado, podemos constatar que o parâmetro k exerce grande influência no tempo de execução.

2 Teorema 2

Teorema 2 (k): o número de números inteiros cujos dígitos pertencem ao conjunto $\{1, 2, ..., m\}$ de k dígitos diferentes é dado pelo produto $m * (m-1) * ... * (m-k+1)$.

Como o número não pode possuir dígitos repetidos, podemos concluir que $m \geq k$.

Caso base ($k = 1$): Para formar um número inteiro de 1 dígito, basta escolher um dos m dígitos do conjunto e este será o número. Sendo assim, podemos formar m números.

Passo indutivo: Seja um número $d_1 d_2 ... d_{k-1}$, de $k-1$ dígitos, com N possibilidades distintas de formação. Inserimos um novo dígito d_k ao final deste, a fim de obtermos um número com k dígitos. Podemos verificar que, dos m dígitos disponíveis para o problema, $k-1$ já foram utilizados até então. Logo, existem $m-(k-1) = m-k+1$ possibilidades para d_k e $N*(m-k+1)$ possibilidades para nosso número de k dígitos. Pela hipótese indutiva, temos que $N = m*(m-1)*...*(m-k+2)$ e, portanto, o conjunto de números formados por k dígitos diferentes, cada dígito dentre m possibilidades, tem $m * (m-1) * ... * (m-k+2) * (m-k+1)$ elementos.

Observação: nos algoritmos desenvolvidos, a fim de generalizar um dígito e permitir um valor arbitrário de m, cada digito é representado por um inteiro de 1 a m. Um número gerado pelo algoritmo, por sua vez, é representado por uma lista de dígitos.

Algoritmo Genérico:

```
função numeros(k, m)

    lista = []

    se k == 1
        para i de 1 até m
            lista.push(i)
        retorna lista
```

```
    listaAnterior = numeros(k-1,m)

    para i de 0 até tamanho(listaAnterior)
        para j de 1 até m
            se j não existe em listaAnterior[i]
                novoNumero <- listaAnterior[i]
                novoNumero.push(j)
                lista.push(novoNumero)

    return lista
```

Implementação em Python:

```
def numbers(k, m):

    newList = []

    # Caso base
    if k == 1:
        for i in range(m):
            newList.append([i+1])
        return newList

    previousList = numbers(k-1,m)

    # para cada numero de k - 1 digitos
    for i in range(len(previousList)):
        # para cada digito entre 0 e m-1
        for j in range(m):
            # se j+1 ainda nao foi utilizado no numero corrente,
            # adicionar ao final deste e salvar na lista
            if (j+1) not in previousList[i]:
                newNumber = list(previousList[i])
                newNumber.append(j+1)
                newList.append(newNumber)

    return newList
```

Exemplo 1: com m=3 e k=1, esse foi o resultado gerado pela implementação em Python:

```
[[1, 2, 3], [1, 2, 4], [1, 3, 2], [1, 3, 4], [1, 4, 2], [1, 4, 3], [2, 1, 3], [2, 1, 4], [2,
3, 1], [2, 3, 4], [2, 4, 1], [2, 4, 3], [3, 1, 2], [3, 1, 4], [3, 2, 1], [3, 2, 4], [3, 4,
1], [3, 4, 2], [4, 1, 2], [4, 1, 3], [4, 2, 1], [4, 2, 3], [4, 3, 1], [4, 3, 2]]
Total: 24
```

Exemplo 2: com m=12 e k=2, esse foi o resultado gerado pela implementação em Python:

```
[[1, 2], [1, 3], [1, 4], [1, 5], [1, 6], [1, 7], [1, 8], [1, 9], [1, 10], [1, 11], [1, 12],
[2, 1], [2, 3], [2, 4], [2, 5], [2, 6], [2, 7], [2, 8], [2, 9], [2, 10], [2, 11], [2, 12],
[3, 1], [3, 2], [3, 4], [3, 5], [3, 6], [3, 7], [3, 8], [3, 9], [3, 10], [3, 11], [3, 12],
[4, 1], [4, 2], [4, 3], [4, 5], [4, 6], [4, 7], [4, 8], [4, 9], [4, 10], [4, 11], [4, 12],
[5, 1], [5, 2], [5, 3], [5, 4], [5, 6], [5, 7], [5, 8], [5, 9], [5, 10], [5, 11], [5, 12],
[6, 1], [6, 2], [6, 3], [6, 4], [6, 5], [6, 7], [6, 8], [6, 9], [6, 10], [6, 11], [6, 12],
```

```
[7, 1], [7, 2], [7, 3], [7, 4], [7, 5], [7, 6], [7, 8], [7, 9], [7, 10], [7, 11], [7, 12],
[8, 1], [8, 2], [8, 3], [8, 4], [8, 5], [8, 6], [8, 7], [8, 9], [8, 10], [8, 11], [8, 12],
[9, 1], [9, 2], [9, 3], [9, 4], [9, 5], [9, 6], [9, 7], [9, 8], [9, 10], [9, 11], [9, 12],
[10, 1], [10, 2], [10, 3], [10, 4], [10, 5], [10, 6], [10, 7], [10, 8], [10, 9], [10, 11],
[10, 12], [11, 1], [11, 2], [11, 3], [11, 4], [11, 5], [11, 6], [11, 7], [11, 8], [11, 9],
[11, 10], [11, 12], [12, 1], [12, 2], [12, 3], [12, 4], [12, 5], [12, 6], [12, 7], [12, 8],
[12, 9], [12, 10], [12, 11]]
Total: 132
```

Testes: A tabela abaixo ilustra os resultados dos testes:

m	k	Execuções	Tempo Total	Tempo/Execução
10	5	102	5.015 s	49.161 ms
10	7	4	6.635 s	1658.823 ms
10	10	1	33.882 s	33882.317 ms
15	5	8	5.184 s	647.980 ms
15	6	1	7.017 s	7016.968 ms
15	7	1	77.290 s	77289.940 ms
20	5	2	6.860 s	3429.841 ms
20	6	1	55.417 s	55416.994 ms
30	5	1	30.049 s	30049.451 ms
100	3	4	5.712 s	1428.066 ms

Conclusão: Podemos verificar que tanto o parâmetro m quanto k exercem grande influência no tempo de execução do programa. Tal fato está diretamente ligado com a implementação utilizada, visto que o aumento de k está relacionado com o aumento das chamadas recursivas, e o aumento de m, com o aumento das iterações. Para os seguintes valores $\{m, k\}$ a computação do algoritmo não foi concluída após mais de 5 minutos de execução: {10,15}, {15,10}, {20,7}, {30,6}, {100,4} e {1000,3}.

3 Teorema 3

Teorema 3 (k): Em um campeonato com $n = 2^k$ equipes, sabe-se construir as $2^k - 1$ rodadas de 2^{k-1} jogos onde cada equipe enfrenta uma equipe diferente em cada rodada.

Intuição: em cada rodada, cada equipe participará de um só jogo, e como cada jogo envolve duas equipes, cada rodada terá $n/2 = 2^{k-1}$ jogos. Como cada equipe só joga com uma outra por rodada, deverão existir $n - 1 = 2^k - 1$ rodadas para que cada equipe possa jogar com todas as outras.

Prova: feita por indução matemática usando k como parâmetro de indução. As equipes serão numeradas de e_1 até $e_{2^k} = e_n$.

Caso base ($k = 1$): temos $n = 2$. Haverá $n - 1 = 1$ rodada, com $n/2 = 1$ jogo. Esse jogo é $[e_1, e_2]$.

Passo indutivo: podemos assumir que o teorema é válido para um certo k (hipótese indutiva). Desejamos provar que, a partir disso, o teorema se torna válido para $k + 1$. Nesse caso, existem 2^{k+1} equipes.

Divide-se as equipes em dois grupos, A e B de igual tamanho: $\{e_1, ..., e_{2^k}\}$ e $\{e_{2^k+1}, ..., e_{2^{k+1}}\}$. Cada grupo tem 2^k equipes. Neste primeiro momento, trataremos os dois grupos como dois torneios independentes e simultâneos. Pela hipótese indutiva, sabemos resolver esses dois problemas (idênticos), e geramos portanto dois torneios, cada um com $2^k - 1$ rodadas de 2^{k-1}. Como os torneios são simultâneos, a rodada 1 do torneio A ocorrerá ao mesmo tempo que a rodada 1 do torneio B, portanto, ao juntar essas rodadas, teremos 2^k jogos por rodada.

Após esse momento inicial, sabemos que cada equipe do grupo A enfrentou todas as outras equipes de seu grupo. O mesmo vale para o grupo B. Resta, portanto, cada equipe do grupo A enfrentar todas as equipes do grupo B, e vice-versa.

Para fazer isso, teremos mais 2^k rodadas de 2^k jogos. Para gerar esses jogos, considere os conjuntos ordenados das equipes do grupo A e do grupo B.

$$a_1 = e_1 \tag{3}$$
$$a_2 = e_2 \tag{4}$$
$$... \tag{5}$$
$$a_{2^k} = e_{2^k} \tag{6}$$
$$\tag{7}$$

$$b_1 = e_{2^k+1} \tag{8}$$
$$b_2 = e_{2^k+2} \tag{9}$$
$$... \tag{10}$$
$$b_{2^k} = e_{2^{k+1}} \tag{11}$$

A primeira rodada deste momento teremos os jogos $[a_1, b_1]$, $[a_2, b_2]$, ..., $[a_{2^k}, b_{2^k}]$

Na segunda rodada, ocorrerá uma rotação nos times de B, de modo que o primeiro time de A enfrentará o último time de B: $[a_1, b_{2^k}]$, $[a_2, b_1]$, ..., $[a_{2^k}, b_{2^k-1}]$

Nas próximas rodadas, continuará ocorrendo essa rotação, até que na 2^k-ésima rodada deste momento teremos: $[a_1, b_2]$, $[a_2, b_3]$, ..., $[a_{2^k-1}, b_{2^k}]$, $[a_{2^k}, b_1]$

O total destes dois momentos é $(2^k - 1) + (2^k) = 2^{k+1} - 1$ rodadas de 2^k jogos, o que está de acordo com o teorema.

Algoritmo genérico:

```
# times é argumento opcional, usado pela recursão
função torneio(k, times)
    se times == nulo
        times = [1, ..., 2^k]

    se k == 1
        jogo <- tupla(times[0], times[1])
        retorna [[jogo]]

    grupo1 <- primeira_metade(times)
    grupo2 <- segunda_metade(times)

    # Momento 1 - divide em dois
    t1 <- torneio(k-1, grupo1)
    t2 <- torneio(k-1, grupo2)
    torneio <- funde_rounds(t1, t2)

    # Momento 2 - ciclo
    para z de 0 ate tamanho(grupo1) - 1
        round <- round_vazio
        para i de 0 ate tamanho(grupo1)
            index1 <- i
            index2 <- (i + z) % tamanho(grupo2)
            jogo <- tupla(grupo1[index1], grupo2[index2])
            round.push(jogo)
```

```
        torneio.push(round)

    retorna torneio
```

Implementação em Python:

```
# start é um argumento opcional, usado pela recursão
# O retorno eh uma lista de rounds
# Um round eh uma lista de jogos
# Um jogo eh uma tupla indicando os dois times
def tournament(k, start = 1):
    # Caso base
    if k == 1:
        return [[(start, start+1)]]

    # MOMENTO 1 - divide em dois

    # Recursao
    t1 = tournament(k-1, start)
    t2 = tournament(k-1, 2**(k-1)+start)

    # Unir os rounds dos dois torneios
    t = []
    for i in range(len(t1)):
        t.append(t1[i] + t2[i])

    # MOMENTO 2 - ciclo

    # Geracao das listas de times
    times1 = range(start, 2**(k-1)+start)
    times2 = range(2**(k-1)+start, 2**(k)+start)

    # z vai ser a variavel que faz o ciclo
    for z in range(len(times1)):
        # Estamos em um round
        round = []
        for i in range(len(times1)):
            # Estamos em um par dentro do ciclo

            # index1 eh simplesmente i
            # index2 muda de modo a fazer o ciclo no segundo grupo
            index1 = i
            index2 = (i + z) % len(times2)

            game = (times1[index1], times2[index2])
            round.append(game)

        # Adicionamos esse round ao conjunto de rounds, t
        t.append(round)

    return t
```

Exemplo: com k=3, esse foi o resultado gerado pela implementação em Python (em anexo, o código usado para printar):

```
Round #01
  Game #01: 1 vs 2
  Game #02: 3 vs 4
  Game #03: 5 vs 6
  Game #04: 7 vs 8
Round #02
  Game #01: 1 vs 3
  Game #02: 2 vs 4
  Game #03: 5 vs 7
  Game #04: 6 vs 8
Round #03
  Game #01: 1 vs 4
  Game #02: 2 vs 3
  Game #03: 5 vs 8
  Game #04: 6 vs 7
Round #04
  Game #01: 1 vs 5
  Game #02: 2 vs 6
  Game #03: 3 vs 7
  Game #04: 4 vs 8
Round #05
  Game #01: 1 vs 6
  Game #02: 2 vs 7
  Game #03: 3 vs 8
  Game #04: 4 vs 5
Round #06
  Game #01: 1 vs 7
  Game #02: 2 vs 8
  Game #03: 3 vs 5
  Game #04: 4 vs 6
Round #07
  Game #01: 1 vs 8
  Game #02: 2 vs 5
  Game #03: 3 vs 6
  Game #04: 4 vs 7
```

Testes: A tabela abaixo ilustra os resultados dos testes. O maior valor de k para o qual o algoritmo gerou as rodadas foi 13.

k	Execuções	Tempo Total	Tempo/Execução
1	7184461	5.000 s	0.000696 ms
2	639199	5.000 s	0.007822 ms
3	180561	5.000 s	0.027692 ms
4	56125	5.000 s	0.089088 ms
5	16534	5.000 s	0.302425 ms
6	4765	5.001 s	1.049445 ms
7	1138	5.004 s	4.397375 ms
8	285	5.016 s	17.600289 ms
9	58	5.026 s	86.647391 ms
10	15	5.331 s	355.405490 ms
11	6	10.625 s	1770.771265 ms
12	6	42.569 s	7094.750086 ms
13	3	85.323 s	28440.937837 ms

Conclusão: o algoritmo derivado da prova indutiva tem várias limitações. Ele não permite um número arbitrário de equipes, somente potências de 2. A sua execução é demorada e exige que sejam feitas muitas chamadas recursivas, da ordem de 2^k. Apesar disso, tem resultados corretos e possui uma explicação interessantíssima. Uma característica dele é que é determinístico: não incorpora nenhum fator de aleatoriedade quanto à geração das rodadas e partidas.