# INF1631 - Estruturas Discretas

## Trabalho 1


PUC
RIO

**Professor: Marcus Poggi**

**Alunos:**

Arthur Cavalcante Gomes Coelho - 1920964

Luiza Del Negro Ciuffo Fernandes - 1721251

# Exercício 1

**(a)**

$P(n-1) : (x^{n-1} - y^{n-1})/(x-y) = 0$

$P(n) \ : \ (x^{n} - y^{n})/(x-y) = 0$

*Caso Base:*

$P(1) : (x^{1} - y^{1})/(x-y) = 0$ ⇒ É válido

$P(2) : (x^{2} - y^{2})/(x-y) = 0$

$P(2) : (x-y) * (x+y)/(x-y) = 0$ ⇒ É válido

*Teorema do Passo Indutivo:*

Provar que $P(n-1) \Rightarrow P(n)$

$P(n) \ : \ (x^{n} - y^{n})/(x-y) = 0$

$P(n) : (x^{n-1} - y^{n-1})(x+y)/(x-y) = 0$

Como assumimos que na hipótese indutiva é verdade, veja que:

$(x^{n-1} - y^{n-1})/(x-y) = 0$

Logo temos que $P(n) : (x^{n-1} - y^{n-1})(x+y)$, que é múltiplo de (x+y)

Então $P(n-1) \Rightarrow P(n)$ ■

**(b)**

```
#include <iostream>
#include <math.h>

using namespace std;

int quocient(long x, long y, int n){
   if (n == 1)
      return ( (x - y) % (x - y) ) == 0;
   return long(pow(x, n-1)) + (y * quocient(x, y, n - 1));
}

int main(){
   cout << quocient(75, 43, 4) << endl;
}
```

# Exercício 2

**(a)**

*Caso Base:*

$n = 1$

Temos um inteiro $x1$ , logo sabemos ordená-lo pois só existe ele.

$n = 2$

Temos dois inteiros, por exemplo $x1 = 89$ e $x2 = 98$ como pegamos o dígito menos significativo primeiro e menor, esse seria o 8 de $x2$ , que é menor que o 9 de $x1$ , logo, $x2$ entraria na frente de $x1$ .

*Passo Indutivo:*

Assumimos que a ordenação está correta para $n - 1$ dígito menos significativos.

Se ordenamos até o *n-ésimo* dígito, o temos ordenados até os *n* dígitos menos significativos uma vez que:

- se tem o *n-ésimo* número distinto, o menor vem antes do maior
- se possuem o *n-ésimo* valor igual, a ordem está correta já que não faz diferença e até esse ponto eles já foram ordenados.

Logo, pela HI, os dois elementos já estavam ordenados até o $n.ésimo - 1$ dígito significativo.

**(b)**

```
sort(array,d){//cada número no array é um número de d digitos
        for i=1 to d do
                int count[10]={0}//inicio
        for j=0 to n do
                count[ndigit of(array[i]) in pass j]++ n//digit é o dígito no lugar j, isso guarda
a contagem dos números
        for k=1 to 10 do
                count[k]=count[k]+count[k-1]
        for j=n-1 até 0 do
                result[count[ndigitof (array[j])]=array[i]//monta o aray final chegando a
posição do array[j] com o contador count[k]
                count[ndigit of (array[j])]--
        for j=0 to n do
                array[j]=result[j]
endfor
end function
} //Assim, o array tem seus números ordenados
```

**(c)**

Sendo $b$ uma base arbitrária, sabemos que seus números vão até seu elemento.

*exemplo:*

$base\ 2\ :\ \{0,1\}$

$base\ 4\ :\ \{0,1,2,3\}$

$base\ 10\ :\ \{0,1,2,3,4,5,6,7,8,9\}$

Dessa forma, conseguimos replicar o mesmo que foi feito anteriormente, uma vez que o dígito menos significativo sempre poderá ser ordenado.

*Caso Base:*

- Base 2:

$n = 1$

Temos um inteiro $x1$, logo sabemos ordená-lo pois só existe ele.

$n = 2$

Temos por exemplo $x1 = 0011$ e $x2 = 0010$ como pegamos o dígito menos significativo primeiro,0 é menor que 1, portanto $x2$ entra antes de $x1$.

- Base 4

$n = 1$

Temos um inteiro $x1$, logo sabemos ordená-lo pois só existe ele.

$n = 2$

Temos por exemplo $x1 = 12$ e $x2 = 21$ como pegamos o dígito menos significativo primeiro,1 é menor que 2, portanto $x2$ entra antes de $x1$.

*Passo Indutivo:*

Assumimos que a ordenação está correta para $n - 1$ dígito menos significativos de base $b$.

Se ordenamos até o *n-ésimo* dígito, o temos ordenados até os $n$ dígitos menos significativos uma vez que:

- se tem o *n-ésimo* número distinto, o menor vem antes do maior
- se possuem o *n-ésimo* valor igual, a ordem está correta já que não faz diferença e até esse ponto eles já foram ordenados.

Logo, pela HI, os dois elementos já estavam ordenados até o *n-ésimo*-1 dígito significativo.

**(d)**

```
sort(array,d,b){//cada número no array é um número de d digitos e base b
        for i=1 to d do
                int count[b]={0}//inicio
        for j=0 to n do
                count[ndigit of(array[i]) in pass j]++// ndigit é o dígito no lugar j, isso guarda
a contagem dos números
        for k=1 to b do
                count[k]=count[k]+count[k-1]
        for j=n-1 até 0 do
                result[count[ndigitof (array[j])]=array[i]// monta o array final chegando a
posição do array[j] com o contador count[k]
                count[ndigit of (array[j])]--
        for j=0 to n do
                array[j]=result[j]
endfor
end function
} //Assim, o array tem seus números ordenados
```

**Experimentação:**

**(a)**

```
using namespace std;

/* counting sort is a subroutine of radix sort that sorts the digits in each place value

i.e: if we're evaluating a 3-digit number in base 10, counting sort will be called to sort the

1's place, then it will be called to sort the 10's place, then it will be called to sort  the

100's place */



    vector<int> countingSort(vector<int> &v,int digit, int base){
        int n = v.size();
        vector<int> res(n);
        // map that counts number of occurrences of the digits in v
        vector<int> occur(base);

        // counts the number of occurences of each digit in A
        for (int i = 0; i < v.size(); i++){
            int digit_of_v_i=  int( v[i] / pow(base, digit) )  % base;
            occur[digit_of_v_i]++;
        }
```

```
    /*  change occur to show the cumulative number of digits up to index */
    for(int i = 1; i < base; i++)
        occur[i] += occur[i-1];

    for(int i = v.size()-1; i >= 0; i--){
        int digit_of_v_i=  int( v[i] / pow(base, digit) )  % base;
        occur[digit_of_v_i]--;
        res[ occur[digit_of_v_i] ] = v[i];
    }

    return res;
}

void radixSort(vector<int> &v, int base){
    // high is the largest number in the vector
    int high = *max_element(v.begin(), v.end()-1);

    // compute the number of digits needed to represent the highest number in v
    int max_digits = floor( ( log(high)/log(base) ) + 1);

    for(int i = 0; i < max_digits; i++)
        v = countingSort(v, i, base);
}


void print_vector(vector<int> &v){
    for (int i = 0; i < v.size(); i++)
        cout << "v[" << i << "] = "  << v[i] << endl;
    cout << "-----------------" << endl;
}

int main(){
    vector<pair<vector<int>, int > > vectors;
    array<int, 9> arr1 = {111, 222, 3333, 444444, 1, 9, 8, 7, 6};
    vector<int> v1(arr1.begin(), arr1.end());
    vectors.push_back(make_pair(v1, 10));


    array<int, 6> arr2 = {0b0111, 0b1001, 0b0001, 0b0011, 0b1011, 0b1111};
    vector<int> v2(arr2.begin(), arr2.end());
    vectors.push_back(make_pair(v2, 2));


    array<int, 6> arr3 = {0xa2, 0xa1, 0xa, 0xbf, 0xc43, 0x01};
    vector<int> v3(arr3.begin(), arr3.end());
    vectors.push_back(make_pair(v3, 16));
```

```
    for(int i = 0; i < vectors.size(); i++){
        vector<int> vit = vectors[i].first;
        int base = vectors[i].second;
        radixSort(vit, base);
        print_vector(vit);
    }

    return 0;
}
```

# Exercício 3

**(a)**

*Caso Base:*

Um grafo denso é aquele que possui o maior número de arestas possível em um grafo não direcionado. Um ciclo Hamiltoniano é aquele que se inicia em um vértice V e passa exatamente uma vez em todos os demais até retornar a seu ponto de partida.
Seja G um grafo denso, e o número de vértices $>= 3$ temos $|V| = 3$ o caso base
Chamamos $\{v1, v2, v3\}$ os vértices de G. Entre eles se ligam arestas de acordo com as regras de um grafo denso. Partindo de um vértice qualquer, por exemplo, $v1$, partindo dele para qualquer outro, por exemplo $v3$ e a partir de $v3$ vamos para $v2$, que pela propriedade, também está conectado. Assim, nos resta a retornar para $v1$, que é o ponto de partida.

*Hipótese Indutiva:*

Seja G=(V,E) um grafo denso, sabemos encontrar um ciclo hamiltoniano onde $|V| = 3$. Nele existem os vértices $v$ e $w$ não adjacentes.

*Passo Indutivo:*

Considerando a Hipótese Indutiva, consideramos achar um grafo hamiltoniano também em um gráfico G', similar a G porém v e w são adjacentes.
De acordo com o enunciado, $d(v) + d(w) \geq n$ .Também acompanhando o que nos foi dado, sabemos que um ciclo começa e termina no mesmo ponto. Consideramos a aresta $(v, w)$, que, se não está presente, estamos lidando com o ciclo G, se não, lidamos com G'. Se temos $n$ como o número de vértices, existem ao menos $n$ arestas, por se tratar de um Ciclo Hamiltoniano, mas como também estamos lidando com um grafo denso, ele possui o maior número de arestas possíveis, se tem a máxima densidade, é um grafo completo e se é um grafo completo, de todos os seus vértices saem arestas que ligam a todos os outros vértices. Dessa forma, vértices vizinhos quaisquer estão conectados a $v$ e $w$. Chamamos, por exemplo, esses dois vértices de $g$ e $h$. Com o que sabemos, podemos escolher vértices como $(v, g)$ e $(w, h)$ e formar um novo ciclo, como: v,g,w,h...v.■

**(b)**
**EM PYTHON( algoritimo completo):**

```
import sys

def read_file(file):
    file_name = file[1]
    mynumbers = []
    with open (file_name, "r") as f:
        for line in f:
            mynumbers.append([int(n) for n in line.strip().split(' ')])
    flag1 = 0
    edge = 0
    vertice = 0
    for pair in mynumbers:
        try:
            x,y = pair[0],pair[1]
            if flag1 == 0:
                edge = x
                vertice = y
                graph = [[0 for w in range(vertice)] for h in range(edge)]
                flag1 = 1
            else:
                graph[x-1][y-1] += 1
            #print
        # Do Something with x and y
        except IndexError:
            print ("A line in the file doesn't have enough entries.")
    #print(graph)
    return graph

def find_all_paths(graph, start, end, path=[]):
    path = path + [start]
    if start == end:
        return [path]
    if start not in graph:
        return None
    paths = []
    for node in graph:
        if node not in path:
            newpaths = find_all_paths(graph, node, end, path)
            for newpath in newpaths:
                paths.append(newpath)
    return paths

def find_cycle(graph):
```

```
    cycles=[]
    for startnode in graph:
        for endnode in graph:
            newpaths = find_all_paths(graph, startnode, endnode)
            for path in newpaths:
                if (len(path)==len(graph)):
                    print (path)
                    cycles.append(path)
    return cycles

graph = read_file(sys.argv)

print ("Finding Hamiltonian Cycles----")
a= find_cycle(graph)
print ("done!")
```

**EM C, uma demonstração de como funciona um ciclo hamiltoniano em uma matriz:**

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <iostream>
#include <assert.h>
#define num 5
using namespace std;

int grafo[num][num] = {
   {0, 1, 0, 1, 0},
   {1, 0, 1, 1, 1},
   {0, 1, 0, 0, 1},
   {1, 1, 0, 0, 1},
   {0, 1, 1, 1, 0},};
 int grafo[num][num] = {
   {0, 1, 0, 1, 0},
   {1, 0, 1, 1, 1},
   {0, 1, 0, 0, 1},
   {1, 1, 0, 0, 0},
   {0, 1, 1, 0, 0},};

int p[num];
void mostra() {// printa na tela o ciclo se hamiltoniano
        cout << "Ciclo: ";

        for (int i = 0; i < num; i++)
                cout << p[i] << " ";
        cout << p[0] << endl;
```

```
}
bool valido(int v, int k) {// verifica se tem vertice, se ja tiver ocupado, para ambos os casos retorna false, se nao, retorna true
	if (grafo[p[k - 1]][v] == 0)
		return false;

	for (int i = 0; i < k; i++)
		if (p[i] == v)
			return false;
	return true;
}
bool achaciclo(int k) {// confere se todos os vertices estao no passeio, verifica se pode realizar um ciclo
	if (k == num) {
		if (grafo[p[k - 1]][p[0]] == 1)
			return true;
		else
			return false;
	}
	for (int v = 1; v < num; v++) {
		if (valido(v, k)) {
			p[k] = v;
			if (achaciclo(k + 1) == true)
				return true;
			p[k] = -1;
		}
	}
	return false;
}
bool hamiltoniano() {
	for (int i = 0; i < num; i++)
		p[i] = -1;
	p[0] = 0; //first vertex as 0

	if (achaciclo(1) == false) {
		cout << "Nao tem solucao" << endl;
		return false;
	}

	mostra();
	return true;
}

int main() {
	hamiltoniano();
}
```

**(c)**
A prova utilizada consistia em sempre encontrar pelo menos um caminho Hamiltoniano, que consiste em um ciclo que sai de um vértice, percorre todos os outros vértices até chegar no mesmo.
Como o número de caminhos hamiltonianos possíveis era muito grande, medimos o código para apenas uma busca de todos os caminhos possíveis.
Não foi possível executar todas as demonstrações devido a demora de execução para encontrar todos os caminhos.

## Especificações:

- Grafo:10_1
  - Number of Hamiltonian Cycles= 3628800
  - Tempo Total: 91.59159994125366 s
  - Tempo Medio: 45795799.97062683 µs
  - Ultima Chamada: 2.86102294921875 µs
  - Estampa 1 do total: 0h 1m 31s 591ms 599µs
  - Estampa 2 do total: 00:01:31.0591
- Grafo:20_1