1 Recursão - Aula 1

1.1 Fatorial

No Ensino Médio, aprendemos a calcular o fatorial de um número n de duas formas:

a)

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 1$$

Exemplos:

$4! = 4 \times 3 \times 2 \times 1 = 24$

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

$7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 5040$

Em Python, a implementação seria:

```python
def fat(n):
    x = 1
    while n > 0:
        x = x * n
        n = n - 1
    return x
```

b)

$$n! = \begin{cases} n \times (n-1)!, & \text{se } n > 0 \\ 1, & \text{caso contrário.} \end{cases}$$

Exemplos:

$1! = 1 \times 0! = 1 \times 1 = 1$

$2! = 2 \times 1! = 2 \times 1 = 2$

$3! = 3 \times 2! = 3 \times 2 = 6$

$4! = 4 \times 3! = 4 \times 6 = 24$

$5! = 5 \times 4! = 5 \times 24 = 120$

Em Python, a implementação seria mais simples e mais fácil de se entender:

```python
def fat(n):
    if n > 0:
        return n * fat(n-1)
    else:
        return 1
```

Recursão consiste em quebrar um problema difícil em problemas menores e cada vez mais simples, até que seja possível resolvê-los. Geralmente, envolve uma função que chama diretamente a si mesma. Permite escrever soluções mais elegantes, legíveis e mais fáceis de programar.

1.2 Soma dos elementos de uma lista

A soma dos elementos de uma lista pode ser expressa da seguinte forma:

```python
def soma(l):
    if len(l) == 0:
        return 0
    else:
        return l[0] + soma(l[1:])
```

Internamente, quando o processador encontra a expressão `fat(3)`, ele resolve-a de seguinte forma:

$$
\begin{aligned}
x &= \underbrace{fat(3)}_{3*fat(2)} \\
&= 3 * \underbrace{fat(2)}_{2*fat(1)} \\
&= 3 * 2 * \underbrace{fat(1)}_{1*fat(0)} \\
&= 3 * 2 * 1 * \underbrace{fat(0)}_{1} \\
&= 3 * 2 * 1 * 1 \\
&= 6
\end{aligned}
$$

Para isso, ele utiliza a pilha de execução, uma parte da memória RAM em que são armazenadas as informações sobre as sub-rotinas ativas no programa. Seu principal uso é registrar o ponto em que cada sub-rotina ativa deve retornar o controle de execução quando termina de executar. Ela não é usada apenas em chamadas recursivas, mas também quando uma função `f(x)` faz chamada a uma outra função `g(x)` qualquer. Da mesma forma, a expressão `soma([1,2,3,4])` seria resolvida da seguinte forma:

$$
\begin{aligned}
s &= \underbrace{soma([1,2,3,4])}_{1+soma([2,3,4])} \\
&= 1 + \underbrace{soma([2,3,4])}_{2+soma([3,4])} \\
&= 1 + 2 + \underbrace{soma([3,4])}_{3+soma([4])} \\
&= 1 + 2 + 3 + \underbrace{soma([4])}_{4+soma([\ ])} \\
&= 1 + 2 + 3 + 4 + \underbrace{soma([\ ])}_{0} \\
&= 1 + 2 + 3 + 4 + 0 \\
&= 10
\end{aligned}
$$

Todos os algoritmos recursivos devem obedecer a três leis importantes:

1. Um algoritmo recursivo deve chamar a si mesmo, recursivamente.
2. Um algoritmo recursivo deve ter ao menos um **caso básico** (condição de parada, ou seja, um problema suficientemente simples de se resolver).
3. Um algoritmo recursivo deve mudar seu estado a cada chamada recursiva, aproximando-se do caso básico.

1.3 Sequência de Fibonacci

A sequência de Fibonacci é um exemplo de algoritmo recursivo que necessita de mais de um caso básico. A sequência é definida por:

$$
\begin{aligned}
F_1 &= 1 \\
F_2 &= 1 \\
F_n &= F_{n-1} + F_{n-2}
\end{aligned}
$$

Assim, define-se que os dois primeiros elementos da sequência são iguais a 1, e cada elemento seguinte é dado pela soma dos **dois** anteriores.

$$
1, 1, \underbrace{2}_{1+1}, \underbrace{3}_{1+2}, \underbrace{5}_{2+3}, \underbrace{8}_{3+5}, \underbrace{13}_{5+8}, \underbrace{21}_{8+13}, \underbrace{34}_{13+21}, \underbrace{55}_{21+34}, \underbrace{89}_{34+55}, \cdots
$$

Em Python, a função de Fibonacci pode ser definida recursivamente da seguinte forma:

```python
def fib(n):
    if n == 1: return 1
    elif n == 2: return 1
    else: return fib(n-1) + fib(n-2)
```

2 Recursão - Aula 2

2.1 Exponenciação

A exponenciação x^n é dada pela seguinte fórmula matemática:

$$x^n = \begin{cases} 1, & \text{se } n == 0 \\ x \cdot x^{n-1}, & \text{caso contrário.} \end{cases}$$

Em Python:

```python
def exp(x, n):
    if n == 0:
        return 1
    else:
        return x * exp(x, n-1)
```

2.2 Maior elemento de uma lista

Para encontrarmos o maior elemento de uma lista:

```python
def maior(x, y):
    if x > y: return x
    else: return y

def maximo(l):
    if len(l) == 1:
        return l[0]
    else:
        return maior( l[0], maximo(l[1:]) )
```

2.3 Máximo Divisor Comum

O Máximo Divisor Comum (MDC) entre dois números pode ser calculado da seguinte forma:

1. $MDC(18, 12)$:

   - Divisores de 18: $\underline{1}$, $\underline{2}$, $\underline{3}$, $\underline{6}$, 9, 18
   - Divisores de 12: $\underline{1}$, $\underline{2}$, $\underline{3}$, 4, $\underline{6}$, 12
   - Como os números sublinhados são os divisores de 18 que também são divisores de 12, o maior deles é 6. Portanto, $MDC(18, 12) = 6$.
   - Podemos chegar no resultado da seguinte forma:
     $18//12 = 1$ (resto da divisão é 6).
     $12//6 = 2$ (resto da divisão é 0). Portanto, 6 é o MDC!

2. $MDC(48, 30)$:

   - Divisores de 48: 1, 2, 3, 4, 6, 8, 12, 16, 24, 48
   - Divisores de 30: 1, 2, 3, 5, 6, 10, 15, 30
   - $MDC(48, 30) = 6$.
   - Da mesma forma que no $MDC(18, 12)$:
     $48//30 = 1$ (resto da divisão é 18).
     $30//18 = 1$ (resto da divisão é 12).
     $18//12 = 1$ (resto da divisão é 6).
     $12//6 = 2$ (resto da divisão é 0). Portanto, 6 é o MDC!

De forma geral:

$$MDC(m, n) = \begin{cases} n, & \text{se } m\%n == 0 \\ MDC(n, m\%n), & \text{caso contrário.} \end{cases}$$

Em Python:

```python
def mdc(m, n):
    if m%n == 0:
        return n
    else:
        return mdc(n, x%n)
```

2.4 Inverter os algarismos de um número

Agora, vamos tentar imprimir os algarismos de um número natural na ordem inversa. Exemplo: $38720 \rightarrow 02783$

$$38720//10 = 3872 \qquad 38720\%10 = 0$$
$$3872//10 = 387 \qquad 3872\%10 = 2$$
$$387//10 = 38 \qquad 387\%10 = 7$$
$$38//10 = 3 \qquad 38\%10 = 8$$
$$3//10 = 0 \qquad 3\%10 = 3$$

A cada passo, o resto da divisão é um número a ser impresso. O algoritmo em Python fica:

```
def inverte(x):
    if x//10 == 0:
        print(x)
    else:
        print(x%10, end='')
        inverte(x//10)
```

3 Recursão - Aula 3

3.1 Torre de Hanoi

Recursão também é utilizada para resolver o problema da Torre de Hanoi.

Figura 1: Torre de Hanoi.

Objetivo:

1. Mover n discos da torre A para a torre B (podendo usar o auxílio da torre C).

Regras:

1. Só é possível mover um disco de cada vez.
2. Um disco nunca pode ficar acima de outro disco menor que ele.

Num grande templo na Índia, há uma placa onde estão fixados três pinos de diamante. Diz a lenda que num deles, no momento da criação, o deus Brahma colocou 64 discos de ouro puro, o maior deles na base e os restantes na ordem decrescente de tamanho até o topo. Os monges deveriam se revezar, transferindo os discos de um pino para outro, obedecendo às regras descritas acima. Quando os 64 discos fossem transferidos do pino em que Deus os colocou para qualquer um dos outros dois, o templo viraria pó e o mundo desapareceria. Veja uma simulação online do jogo em `https://www.matematica.pt/fun/hanoi.php` .

A solução não é difícil, mas trabalhosa. Para $n = 3$, o número mínimo de movimentos é 7. Exemplo:

$$
\begin{aligned}
A &\rightarrow B \\
A &\rightarrow C \\
B &\rightarrow C \\
A &\rightarrow B \\
C &\rightarrow A \\
C &\rightarrow B \\
A &\rightarrow B
\end{aligned}
$$

Para $n = 4$, o número mínimo de movimentos é 15. Exemplo:

$$A \to C$$
$$A \to V$$
$$C \to V$$
$$A \to C$$
$$B \to A$$
$$B \to C$$
$$A \to C$$
$$A \to B$$
$$C \to B$$
$$C \to A$$
$$B \to A$$
$$C \to B$$
$$A \to C$$
$$A \to B$$
$$C \to B$$

Para n discos, o menor número de passos é $2^n - 1$. Se $n = 64$ (como na lenda do Deus Brahma), o número mínimo de movimentos é 18.446.744.073.709.551.615. Mesmo que os monges fizessem um movimento por segundo, sem descanso, levariam 585.000.000.000 anos.

Apesar de trabalhoso, o algoritmo é simples:

```python
def hanoi(n, origem, destino, auxiliar):
    if n == 1:
        print(origem, "->", destino)
    else:
        hanoi(n-1, origem, auxiliar, destino)
        print(origem, "->", destino)
        hanoi(n-1, auxiliar, destino, origem)
```

4 Recursão - Aula 4

4.1 Permutações

Permutações de uma lista com 1 elemento:

$$perms([1]) = [1]$$

Permutações de uma lista com 2 elementos:

$$
\begin{aligned}
perms([1,2]) = &[1,2]\\
&[2,1]
\end{aligned}
$$

Permutações de uma lista com 3 elementos:

$$
\begin{aligned}
perms([1,2,3]) = &[\textcolor{red}{1},2,3]\\
&[\textcolor{red}{1},3,2]\\
&[\textcolor{green}{2},1,3]\\
&[\textcolor{green}{2},3,1]\\
&[\textcolor{blue}{3},1,2]\\
&[\textcolor{blue}{3},2,1]
\end{aligned}
$$

Permutações de uma lista com 4 elementos:

$$
\begin{aligned}
perms([1,2,3,4]) = &[1,2,3,4]\\
&[1,2,4,3]\\
&[1,3,2,4]\\
&[1,3,4,2]\\
&[1,4,2,3]\\
&[1,4,3,2]\\
&[2,1,3,4]\\
&[2,1,4,3]\\
&[2,3,1,4]\\
&[2,3,4,1]\\
&[2,4,1,3]\\
&[2,4,3,1]\\
&[3,1,2,4]\\
&[3,1,4,2]\\
&[3,2,1,4]\\
&[3,2,4,1]\\
&[3,4,1,2]\\
&[3,4,2,1]\\
&[4,1,2,3]\\
&[4,1,3,2]\\
&[4,2,1,3]\\
&[4,2,3,1]\\
&[4,3,1,2]\\
&[4,3,2,1]
\end{aligned}
$$

De forma geral, para imprimir as permutações de n elementos, devemos:

- Fixar o primeiro elemento.
- Gerar as permutações do restante da lista, recursivamente, e imprimir o resultado.
- Trocar o primeiro elemento por outro e voltar ao passo 1, até que todos os elementos tenham sido colocados na primeira posição

```python
def troca(l, i, j):
    aux = l[i]
    l[i] = l[j]
```

```python
    l[j] = aux

def perms(l, pos=0):
    if pos == len(l)-1:
        print(l)
    else:
        for i in range(pos, len(l)):
            troca(l, pos, i)
            perms(l, pos+1)
            troca(l, pos, i)


def main(args):
    perms( [1,2,3,4] )

if __name__ == '__main__':
    import sys
    sys.exit(main(sys.argv))
```

Adaptando a função para que retorne uma lista com todas as permutações, ao invés de imprimi-las na tela:

```python
def troca(l, i, j):
    aux = l[i]
    l[i] = l[j]
    l[j] = aux

def perms(l, pos=0):
    if pos == len(l)-1:
        return [l.copy()]
    else:
        result = []
        for i in range(pos, len(l)):
            troca(l, pos, i)
            result += perms(l, pos+1)
            troca(l, pos, i)
        return result

def main(args):
    l = perms( [1,2,3,4] )
    print(l)

if __name__ == '__main__':
```

```
    import sys
    sys.exit(main(sys.argv))
```

5 Recursão - Extra

5.1 TSP - Solução Ótima

No **Problema do Caixeiro Viajante** (do inglês, Travelling Salesman Problem, ou apenas **TSP**), um caixeiro viajante precisa visitar vários locais, partindo de um local inicial qualquer, passando por todos os locais exatamente uma vez e voltando ao local inicial no fim do percurso. O caixeiro deve fazer esse caminho de forma que a distância total percorrida seja a menor possível. Chamamos uma rota, ou caminho, de **tour**. Chamamos o caminho de menor custo possível de **tour ótimo**. O TSP é fácil de ser resolvido: geram-se todas as permutações de locais possíveis e calcula-se a distância total de cada uma delas. A permutação com a menor distância total é o *tour* procurado.

Vamos considerar que cada local é representado por uma tupla contendo o nome de uma cidade e sua posição no plano cartesiano. Exemplo:

```
cidades = [ ("A", (0,4)), ("B", (1,0)), ("C", (1,1)), ("D", (2,2)),
            ("E", (3,2)), ("F", (3,3)), ("G", (4,1)) ]
```

Vamos considerar também que a distância entre dois pontos quaisquer é a Euclidiana, dada por:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

O código do TSP utilizando permutações é:

```
import random, matplotlib.pyplot as plt

'''
Funcoes para visualizar um caminho do TSP na tela
'''

def showTSP(caminho, titulo):
    xs = [coords[0] for (cidade, coords) in caminho]
    ys = [coords[1] for (cidade, coords) in caminho]
    nomes = [cidade for (cidade, coords) in caminho]

    for i in range(len(xs)):
        plt.annotate(nomes[i], # this is the text
                     (xs[i],ys[i]), # this is the point to label
```

```
                        textcoords="offset points", # how to positio
                        xytext=(5,5), # distance from text to points
                        ha='center') # horizontal alignment can be l

    plt.plot(xs+[xs[0]], ys+[ys[0]], 'pb-')
    plt.plot([xs[0]], [ys[0]], 'pb-', color='red')

    t = "{} (Custo: {:.2f})".format(titulo, custo(caminho))
    plt.suptitle(t)
    plt.gca().set_aspect('equal', adjustable='box')

    plt.show()

def show2OPT(caminho, A, B):
    xs = [coords[0] for (cidade, coords) in caminho]
    ys = [coords[1] for (cidade, coords) in caminho]
    nomes = [cidade for (cidade, coords) in caminho]

    for i in range(len(xs)):
        plt.annotate(nomes[i], # this is the text
                        (xs[i],ys[i]), # this is the point to label
                        textcoords="offset points", # how to positio
                        xytext=(5,5), # distance from text to points
                        ha='center') # horizontal alignment can be l

    plt.plot(xs, ys, 'pb-')
    plt.plot([xs[0]], [ys[0]], 'pb-', color='red')

    xs = (caminho[A][1][0], caminho[A+1][1][0])
    ys = (caminho[A][1][1], caminho[A+1][1][1])
    plt.plot(xs, ys, 'pb-', color='red')

    xs = (caminho[B][1][0], caminho[B+1][1][0])
    ys = (caminho[B][1][1], caminho[B+1][1][1])
    plt.plot(xs, ys, 'pb-', color='red')

    xs = (caminho[A][1][0], caminho[B][1][0])
    ys = (caminho[A][1][1], caminho[B][1][1])
    plt.plot(xs, ys, 'pb--', color='gray')

    xs = (caminho[A+1][1][0], caminho[B+1][1][0])
    ys = (caminho[A+1][1][1], caminho[B+1][1][1])
```

```
    plt.plot(xs, ys, 'pb--', color='gray')

    t = "Trocando arestas (Custo atual: {:.2f})".format(custo(caminh
    plt.suptitle(t)
    plt.gca().set_aspect('equal', adjustable='box')

    plt.show()

'''
Permutacoes
'''

def troca(l, i, j):
    aux = l[i]
    l[i] = l[j]
    l[j] = aux

def perms(l, pos =0):
    if pos == len(l)-1:
        return [l.copy ()]
    else:
        result = []
        for i in range(pos, len(l)):
            troca(l, pos, i)
            result += perms(l, pos +1)
            troca(l, pos, i)
        return result

'''
Funcoes auxiliares para o TSP
'''

def dist(p1, p2):
    x1, y1 = p1
    x2, y2 = p2

    return ( (x1-x2 )**2 + (y1-y2 )**2 ) ** 0.5

def custo(rota, completo=False):
    if len(rota) == 1: return 0

    if completo: c = 0
```

```python
    else: c = c = dist(rota [0][1], rota[-1][1])

    for i in range(len(rota)-1):
        c += dist(rota[i][1], rota[i+1][1])
    return c

def printRota (rota ):
    for cidade in rota:
        print(cidade[0], end=" ")
    print(rota[0][0], end=" ")
    print("- custo: {:.2f}".format(custo(rota )))

'''
Vizinho Mais Pr ximo
'''

def melhorRota (rotas ):
    melhor = rotas[0]

    for rota in rotas[1:]:
        if custo(rota) < custo(melhor):
            melhor = rota

    return melhor

def tsp(cidades):
    rotas = perms(cidades, 1)
    melhor = melhorRota (rotas)
    printRota (melhor)
    return melhor

def nnIt(cidades):
    for pos in range(0, len(cidades)-1):
        maisProx = pos+1
        for i in range(pos+2, len(cidades)):
            d1 = dist(cidades[pos][1], cidades[maisProx][1])
            d2 = dist(cidades[pos][1], cidades[i][1])

            if d2 < d1:
                maisProx = i

        if maisProx != pos+1: troca(cidades, maisProx, pos+1)
```

```

def nnRec(cidades, pos=0):
    if pos < len(cidades)-1:
        maisProx = pos+1
        for i in range(pos+2, len(cidades)):
            d1 = dist(cidades[pos][1], cidades[maisProx][1])
            d2 = dist(cidades[pos][1], cidades[i][1])

            if d2 < d1:
                maisProx = i

        if maisProx != pos+1: troca(cidades, maisProx, pos+1)

        nnRec(cidades, pos+1)

def nn(cidades):
    l = cidades.copy()
    nnIt(l)
    printRota(l)
    return l

'''
2-OPT
'''

def dois_opt(rota, show=True):
    rota = rota + [rota[0]]
    melhor = rota
    melhorou = True

    while melhorou:
        melhorou = False

        for A in range(0, len(rota)-3):
            for B in range(A+2, len(rota)-1):

                if A==0 and B==(len(rota)-2): continue

                novaRota = rota[:A+1] + rota[B:A:-1] + rota[B+1:]

                if custo(novaRota, True) < custo(melhor, True):
                    if show: show2OPT(rota, A, B)
```

```
                    melhor = novaRota
                    melhorou = True
                    if show: showTSP(novaRota, "Caminho ap s troca"

        rota = melhor
        if show: showTSP(rota, "Novo caminho")

    printRota(rota)
    return rota

'''
Funcao para gerar cidades aleatorias
'''

def gerarCidades():
    cidades = []
    n = int(input("Digite o n mero de cidades: "))

    for i in range(n):
        nome = chr(i+ord('A'))
        x = random.randint(0, 400)
        y = random.randint(0, 300)

        cidades.append( (nome, (x, y)) )

    return cidades

def main(args):

    cidades = [ ("A", (0,4)), ("B", (1,0)), ("C", (1,1)), ("D", (2,2
                    ("E", (3,2)), ("F", (3,3)), ("G", (4,1)) ]

    cidades = gerarCidades()
    printRota (cidades)

    showTSP(cidades, "Ordem Inicial")

    outra = nn(cidades)
    showTSP(outra, "Vizinho Mais Pr ximo")

    melhorada = dois_opt(outra, show=False)
    showTSP(melhorada, "Resultado 2-OPT")
```

```python

    return 0

if __name__ == '__main__':
    import sys
    sys.exit(main(sys.argv))
```

Esta seria uma ótima solução, se não fosse por um detalhe: dados N locais, são geradas $N!$ permutações, o que torna esta solução intratável do ponto de vista computacional. Basta observar que para se obter uma solução de rota com 21 cidades, seriam explorados

$$21! = 51.090.942.171.709.440.000$$

tours diferentes. Supondo que se tem um processador de 3.6GHz, que pode fazer 3.6×10^9 ciclos por segundo, e que pode gerar um *tour* a cada ciclo do processador, seriam levados

$$51.090.942.171.709.440.000 \div 3.600.000.000 = 14.191.928.382$$

segundos para se encontrar e explorar todos os caminhos possíveis. Mas 14.191.928.382 segundos são aproximadamente 450 anos! E não se deseja esperar 450 anos para encontrar o melhor *tour* entre apenas 21 locais. Devido a essa alta complexidade, existem diversos algoritmos que encontram soluções para o problema num tempo computacional satisfatório, mas nenhuma delas garante que a solução encontrada é a ótima.

5.2 Vizinho Mais Próximo

Um dos algoritmos mais famosos para encontrar uma solução para o TSP num tempo satisfatório é o "Vizinho Mais Próximo" (Nearest Neighbour, ou **NN**). No algoritmo NN, o *tour* se inicia com um local qualquer. Enquanto não se insere todos os locais, deve ser escolhido o destino mais próximo do último local inserido, dentre todos os demais locais que ainda não estão no *tour* – daí o nome vizinho mais próximo. Esse destino mais próximo é então inserido no *tour*. Quando não houver mais locais a serem inseridos, o algoritmo acaba.

Considere os quatro locais a serem visitados, mostrados no plano abaixo:

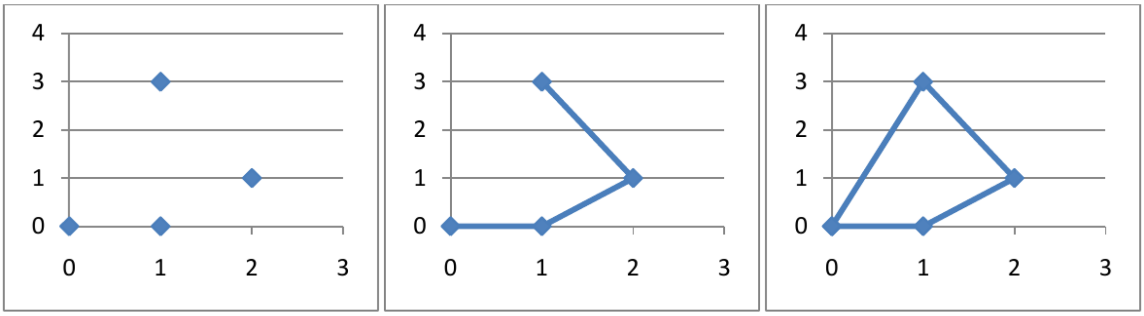

Figura 2: Passos do algoritmo Vizinho Mais Próximo.

A primeira parte da figura mostra a representação dos pontos $(0.0, 0.0)$, $(1.0, 3.0)$, $(2.0, 1.0)$ e $(1.0, 0.0)$, que devem ser visitados pelo caixeiro. A segunda parte mostra a ordem em que ele visitará esses locais. A terceira mostra o caminho total, incluindo a volta ao local inicial no fim do percurso.

O primeiro passo do NN seria escolher o local inicial. No caso do exemplo, o local de partida é o ponto $(0.0, 0.0)$. Em seguida, é inserido no *tour* o local mais perto do último que foi inserido. No exemplo, o local mais perto de $(0.0, 0.0)$ está no ponto $(1.0, 0.0)$. O próximo a ser inserido está no ponto $(2.0, 1.0)$, e assim sucessivamente.

Vamos considerar que cada local é representado por uma tupla contendo o nome de uma cidade e sua posição no plano cartesiano. Exemplo:

```
cidades = [ ("A", (0,4)), ("B", (1,0)), ("C", (1,1)), ("D", (2,2)),
            ("E", (3,2)), ("F", (3,3)), ("G", (4,1)) ]
```

Vamos considerar também que a distância entre dois pontos quaisquer é a Euclidiana, dada por:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Além disso, vamos considerar que o primeiro local da lista será o ponto de partida do caixeiro. No momento inicial, portanto, o *tour* já estará corretamente definido até a posição 0 da lista.

```python
from tsp import dist, custo, printRota
from perms import troca

def nn(cidades):
    for pos in range(len(cidades)-2):
        maisProx = pos+1
        for i in range(pos+2, len(cidades)):
            d1 = dist(cidades[pos][1], cidades[maisProx][1])
            d2 = dist(cidades[pos][1], cidades[i][1])
            if d2 < d1:
```

```
                maisProx = i

        if maisProx != pos+1: troca(cidades, maisProx, pos+1)


cidades = [ ("A", (0,4)), ("B", (1,0)), ("C", (1,1)),
            ("D", (2,2)), ("F", (3,3)), ("G", (4,1)) ]
nn(cidades)
printRota(cidades)
```

```
from tsp import dist, custo, printRota
from perms import troca
from random import randint
from nn import nn

def nnRec(cidades, pos=0):
    if pos < len(cidades)-2:
        maisProx = pos+1
        for i in range(pos+2, len(cidades)):
            d1 = dist(cidades[pos][1], cidades[maisProx][1])
            d2 = dist(cidades[pos][1], cidades[i][1])
            if d2 < d1:
                maisProx = i

        if maisProx != pos+1: troca(cidades, maisProx, pos+1)
        nnRec(cidades, pos+1)

def main(args):
    cidades = []
    for i in range(30):
        cidades.append( (i+1, (randint(0,20),randint(0,20))) )

    cidades2 = cidades[:]

    nn(cidades)
    printRota(cidades)

    nnRec(cidades2)
    printRota(cidades2)


if __name__ == '__main__':
```

```
    import sys
    sys.exit(main(sys.argv))
```