- # Biblioteca
    - Decimal
        - from decimal import Decimal
          ganhos_do_mes = Decimal('99.91') * 5
          print(ganhos_do_mes) #
          gastos_do_mes = Decimal('110.1') * 3
          print(gastos_do_mes)
    - Copy
        - import copy
          y = [[1,2,3], [4,5,6]]
          x = copy.deepcopy(y)
          x[0].append(10)
          print(y) # [[1, 2, 3], [4, 5, 6]]
          print(x) # [[1, 2, 3, 10], [4, 5, 6]]

- # Ordenação

Quando usar

Cada algoritmo de ordenação tem características específicas que o tornam mais adequado para diferentes tipos de problemas. A escolha do algoritmo de ordenação depende de vários fatores, como o tamanho do conjunto de dados, a necessidade de otimização de tempo ou espaço e a natureza dos dados. Aqui está um guia para quando usar alguns dos algoritmos mais comuns:

1. Bubble Sort

- **Quando usar**: Quase nunca é usado em prática, mas pode ser útil em cenários de aprendizado ou quando o conjunto de dados é muito pequeno.
- **Complexidade**: $O(n^2)$
- **Características**: Simples de implementar, mas ineficiente para grandes conjuntos de dados. Pode ser útil se o conjunto de dados já estiver quase ordenado.

2. Selection Sort

- **Quando usar**: Quando o custo de troca (swap) é muito alto e a memória é uma restrição maior que o tempo.
- **Complexidade**: $O(n^2)$

- **Características**: Sempre realiza o mesmo número de comparações independentemente da ordenação inicial dos dados. Não é estável e é lento em conjuntos grandes.

3. Insertion Sort

- **Quando usar**: Útil para conjuntos pequenos ou parcialmente ordenados, como quando você precisa inserir novos elementos em uma lista já ordenada.
- **Complexidade**: $O(n^2)$ no pior caso, mas O(n) se o conjunto de dados estiver quase ordenado.
- **Características**: Muito eficiente para listas pequenas ou quase ordenadas. Também é estável (preserva a ordem de elementos iguais).

4. Merge Sort

- **Quando usar**: Quando você precisa de um algoritmo estável e com complexidade garantida de O(n log n), especialmente em casos de dados muito grandes que não cabem na memória (divide and conquer).
- **Complexidade**: O(n log n)
- **Características**: Estável, funciona bem em dados grandes, mas exige memória extra para fazer a divisão. Utilizado em muitos algoritmos de bibliotecas padrão.

5. Quick Sort

- **Quando usar**: Geralmente uma escolha padrão quando você quer uma boa eficiência prática e o conjunto de dados cabe na memória.
- **Complexidade**: O(n log n) em média, mas pode ser $O(n^2)$ no pior caso.
- **Características**: Muito rápido em muitos casos, mas é instável e pode ter problemas de desempenho em casos extremos (como listas já ordenadas ou inversamente ordenadas, sem otimizações).

6. Heap Sort

- **Quando usar**: Quando você precisa de eficiência de tempo garantida em O(n log n) e quer um algoritmo que funcione **in-place** (sem usar muita memória extra).
- **Complexidade**: O(n log n)
- **Características**: Não é estável, mas é eficiente em tempo e não precisa de memória adicional significativa.

7. Counting Sort

- **Quando usar**: Quando você tem um conjunto de dados com um intervalo limitado de valores (como números inteiros) e deseja um tempo de execução linear.
- **Complexidade**: O(n + k), onde *k* é o valor máximo no conjunto de dados.
- **Características**: Muito rápido para conjuntos de dados com um intervalo pequeno, mas não é prático para grandes intervalos de valores. Também requer memória extra.

8. Radix Sort

- **Quando usar**: Quando os dados são números inteiros ou strings e você precisa de um algoritmo de tempo linear. Funciona bem para números com tamanho fixo de bits ou dígitos.
- **Complexidade**: O(nk), onde *k* é o número de dígitos ou bits.
- **Características**: Não é in-place, mas pode ser muito rápido em conjuntos de dados específicos, como inteiros ou strings com comprimento limitado.

9. Shell Sort

- **Quando usar**: Quando você quer uma versão mais eficiente do Insertion Sort para listas maiores.
- **Complexidade**: Varia com a escolha da sequência de incremento, mas pode ser O(n log n) com as melhores sequências.
- **Características**: Uma melhoria do Insertion Sort, útil em listas maiores, mas ainda não é garantido ser O(n log n) em todos os casos.

Resumo:

- **Conjuntos pequenos ou quase ordenados**: Use **Insertion Sort**.
- **Conjuntos grandes** com boa eficiência prática: Use **Quick Sort** ou **Merge Sort**.
- **Memória limitada** e você quer tempo garantido: Use **Heap Sort**.
- **Valores inteiros com intervalo limitado**: Use **Counting Sort** ou **Radix Sort**.
- **Aprendizado ou casos simples**: Use **Bubble Sort** ou **Selection Sort** (embora geralmente não sejam eficientes).

Esses fatores ajudam a decidir qual algoritmo de ordenação usar em diferentes cenários.

Quicksort

O QuickSort é um algoritmo de ordenação que escolhe um elemento como "pivô" e, a partir desse pivô, divide a lista em duas partes: uma com valores menores e outra com valores maiores. Depois, ele ordena essas duas partes de forma recursiva.

Código exemplo:

```
def quicksort(arr):
  if len(arr) <= 1:
    return arr
  else:
    pivot = arr[0]
    left = [x for x in arr[1:] if x < pivot]
    right = [x for x in arr[1:] if x >= pivot]
    return quicksort(left) + [pivot] + quicksort(right)
```

MergeSort

O MergeSort divide a lista ao meio, ordena cada metade separadamente e depois junta (merge) as duas partes já ordenadas. É um algoritmo eficiente para grandes listas.

Código exemplo:

```
def mergesort(arr):
    if len(arr) > 1:
    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]

    mergesort(left)
    mergesort(right)

    i = j = k = 0

    while i < len(left) and j < len(right):
    if left[i] < right[j]:
    arr[k] = left[i]
    i += 1
```

```
        else:
        arr[k] = right[j]
        j += 1
        k += 1

        while i < len(left):
        arr[k] = left[i]
        i += 1
            k += 1

        while j < len(right):
        arr[k] = right[j]
        j += 1
        k += 1

arr = [12, 11, 13, 5, 6, 7]
mergesort(arr)
print(arr)  # Saída: [5, 6, 7, 11, 12, 13]
```

HeapSort

O HeapSort organiza os elementos usando uma estrutura chamada heap, removendo os menores ou maiores elementos e construindo uma lista ordenada.

Código exemplo:

```
import heapq

def heapsort(arr):
        heapq.heapify(arr)
        sorted_arr = [heapq.heappop(arr) for _ in range(len(arr))]
        return sorted_arr

arr = [12, 11, 13, 5, 6, 7]
print(heapsort(arr))  # Saída: [5, 6, 7, 11, 12, 13]
```

Counting Sort

O Counting Sort conta a quantidade de vezes que cada valor aparece e usa essa contagem para ordenar a lista. É mais eficiente quando os números têm um intervalo pequeno.

Código exemplo:

```
def counting_sort(arr):
        max_value = max(arr)
        count = [0] * (max_value + 1)
        for num in arr:
        count[num] += 1
        sorted_arr = []
        for i, c in enumerate(count):
        sorted_arr.extend([i] * c)
        return sorted_arr

arr = [1, 4, 1, 2, 7, 5, 2]
print(counting_sort(arr))  # Saída: [1, 1, 2, 2, 4, 5, 7]
```

Radix Sort

O Radix Sort ordena os números por cada dígito, começando pelo menos significativo, usando outro algoritmo de ordenação para cada posição.

Código exemplo:

```
def counting_sort_for_radix(arr, exp):
        n = len(arr)
        output = [0] * n
        count = [0] * 10

        for i in range(n):
        index = (arr[i] // exp) % 10
        count[index] += 1

        for i in range(1, 10):
```

```
        count[i] += count[i - 1]

        i = n - 1
        while i >= 0:
        index = (arr[i] // exp) % 10
        output[count[index] - 1] = arr[i]
        count[index] -= 1
        i -= 1

        for i in range(n):
        arr[i] = output[i]

def radix_sort(arr):
        max_value = max(arr)
        exp = 1
   while max_value // exp > 0:
        counting_sort_for_radix(arr, exp)
        exp *= 10

arr = [170, 45, 75, 90, 802, 24, 2, 66]
radix_sort(arr)
print(arr)  # Saída: [2, 24, 45, 66, 75, 90, 170, 802]
```

Bubble Sort

O Bubble Sort compara repetidamente elementos adjacentes e os troca se estiverem na ordem errada. É o mais simples, mas também o mais lento.

Código exemplo:

```
def bubble_sort(arr):
        n = len(arr)
        for i in range(n):
        for j in range(0, n-i-1):
        if arr[j] > arr[j+1]:
        arr[j], arr[j+1] = arr[j+1], arr[j]

arr = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(arr)
print(arr)  # Saída: [11, 12, 22, 25, 34, 64, 90]
```

Recursividade

Definição: A recursividade é uma técnica de programação onde uma função se chama novamente durante sua execução para resolver um problema. É especialmente útil quando um problema pode ser dividido em subproblemas menores e semelhantes. Para que a recursão funcione, é importante definir um *caso base*, que é uma condição que encerra a recursão, evitando que a função chame a si mesma indefinidamente.

Códigos de exemplo:

- Problema de Joséphus
- Museu (quantidade de figuras na pintura)

Exemplo: Vamos usar a função de Fibonacci, que é um clássico exemplo de recursividade. A sequência de Fibonacci é uma série de números onde cada número é a soma dos dois anteriores, começando de 0 e 1.

A sequência é:
0, 1, 1, 2, 3, 5, 8, 13, ...

```
def fibonacci(n):
    # Caso base: se n é 0 ou 1, retorna n
    if n <= 0:
    return 0
    elif n == 1:
    return 1
    else:
    # Chamada recursiva
    return fibonacci(n - 1) + fibonacci(n - 2)


# Testando a função
numero = 6
```

```
resultado = fibonacci(numero)

print(f"O {numero}º número da sequência de Fibonacci é: {resultado}")
```

Matriz

Uma matriz é uma coleção bidimensional de números organizados em linhas e colunas. Em programação, matrizes são frequentemente utilizadas para representar dados, realizar operações matemáticas e armazenar informações de forma estruturada.

Código exemplo:

```
# Criação de uma matriz 4x4 inicializada com zeros

matriz = [[0 for i in range(4)] for j in range(4)]  # Compreensão de lista para criar uma matriz

count = 0  # Contador para preencher a matriz


# Preenchendo a matriz com números sequenciais

for linha in range(4):  # Loop sobre cada linha

        for coluna in range(4):  # Loop sobre cada coluna

        matriz[linha][coluna] = count  # Atribui o valor do contador à posição atual

        count += 1  # Incrementa o contador


# Exibindo a matriz formatada

for linha in range(4):  # Loop sobre cada linha para impressão

        for coluna in range(4):  # Loop sobre cada coluna na linha atual

        print("%4d" % matriz[linha][coluna], end='')  # Imprime o elemento com formatação

        print()  # Nova linha após imprimir todos os elementos da linha
```

Conversões

Conversões de tipos de dados em Python.

Códigos exemplo:

```
# 1. String para Inteiro
numero_str = "123"
numero_int = int(numero_str)
print(f"String convertida para inteiro: {numero_int}")

# 2. String para Float
numero_float_str = "123.45"
numero_float = float(numero_float_str)
print(f"String convertida para float: {numero_float}")

# 3. Inteiro para String
numero = 456
numero_str = str(numero)
print(f"Inteiro convertido para string: '{numero_str}'")

# 4. Float para String
numero_float = 78.90
numero_float_str = str(numero_float)
print(f"Float convertido para string: '{numero_float_str}'")

# 5. Lista para Conjunto
lista = [1, 2, 2, 3, 4, 4]
conjunto = set(lista)
print(f"Lista convertida para conjunto: {conjunto}")
```

```
# 6. Conjunto para Lista
conjunto = {1, 2, 3, 4}
lista_do_conjunto = list(conjunto)
print(f"Conjunto convertido para lista: {lista_do_conjunto}")

# 7. Lista de Strings para Lista de Inteiros
lista_str = ["1", "2", "3"]
lista_int = list(map(int, lista_str))
print(f"Lista de strings convertida para lista de inteiros: {lista_int}")

# 8. Dicionário para Lista de Tuplas
dicionario = {'a': 1, 'b': 2, 'c': 3}
lista_tuplas = list(dicionario.items())
print(f"Dicionário convertido para lista de tuplas: {lista_tuplas}")

# 9. String para Lista
string = "hello"
lista_de_caracteres = list(string)
print(f"String convertida para lista de caracteres: {lista_de_caracteres}")

# 10. Tupla para Lista
tupla = (1, 2, 3)
lista_da_tupla = list(tupla)
print(f"Tupla convertida para lista: {lista_da_tupla}")
```

Busca

Busca binária

É um método para encontrar um número em uma lista que já está ordenada. O algoritmo olha para o número do meio da lista e decide se o número que você quer está antes ou depois dele. Assim, ele corta pela metade a lista até encontrar o número ou saber que ele não está lá.

Código exemplo:

```
# Algoritmo de Busca Binária


def busca_binaria(arr, x):
    """
    Realiza busca binária em um array ordenado.
    :param arr: lista ordenada onde procurar
    :param x: valor a ser encontrado
    :return: índice do valor se encontrado, caso contrário -1
    """
    esquerda, direita = 0, len(arr) - 1
    while esquerda <= direita:
    meio = (esquerda + direita) // 2  # Calcula o índice do meio
    if arr[meio] == x:  # Valor encontrado
    return meio
    elif arr[meio] < x:  # Procura na metade direita
    esquerda = meio + 1
    else:  # Procura na metade esquerda
    direita = meio - 1
    return -1  # Valor não encontrado
```

```
# Testando a busca binária
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]
valor_buscado = 5
resultado_binaria = busca_binaria(arr, valor_buscado)
print(f"Busca Binária: O valor {valor_buscado} está no índice {resultado_binaria}.")
```

Busca em largura (BFS)

É uma maneira de explorar um grafo começando de um ponto. O algoritmo visita todos os vizinhos desse ponto primeiro antes de ir para os vizinhos dos vizinhos. Ele usa uma fila para saber qual nó visitar a seguir.

Código exemplo:

```
# Algoritmo de Busca em Largura (BFS)

from collections import deque

def busca_em_largura(grafo, inicio):
    """
    Realiza busca em largura em um grafo.
    :param grafo: dicionário representando o grafo
    :param inicio: nó onde começar a busca
    :return: lista dos nós visitados na ordem
    """
    visitados = []
    fila = deque([inicio])  # Inicializa a fila

    while fila:
    vertice = fila.popleft()  # Remove o primeiro elemento da fila
```

```
        if vertice not in visitados:
            visitados.append(vertice)  # Marca como visitado
            fila.extend(neighbors for neighbors in grafo[vertice] if neighbors not in visitados)  # Adiciona vizinhos à fila

        return visitados

# Testando a busca em largura
grafo = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
resultado_bfs = busca_em_largura(grafo, 'A')
print(f"Busca em Largura: Nós visitados na ordem: {resultado_bfs}")
```

Busca em profundidade (DFS)

É outra forma de explorar um grafo. O algoritmo vai o mais longe possível em um caminho antes de voltar e tentar outro caminho. Ele geralmente usa uma pilha (ou recursão) para lembrar por onde já passou.

Código exemplo:

```
# Algoritmo de Busca em Profundidade (DFS)

def busca_em_profundidade(grafo, inicio, visitados=None):
```

```
    """
    Realiza busca em profundidade em um grafo.
    :param grafo: dicionário representando o grafo
    :param inicio: nó onde começar a busca
    :param visitados: conjunto de nós visitados
    :return: lista dos nós visitados na ordem
    """
    if visitados is None:
    visitados = []  # Inicializa a lista de visitados
    visitados.append(inicio)  # Marca o nó atual como visitado
    for vizinho in grafo[inicio]:
    if vizinho not in visitados:
    busca_em_profundidade(grafo, vizinho, visitados)  # Chamada recursiva para o vizinho
    return visitados

# Testando a busca em profundidade
resultado_dfs = busca_em_profundidade(grafo, 'A')
print(f"Busca em Profundidade: Nós visitados na ordem: {resultado_dfs}")
```

Grafos

Algoritmo de Dijkstra (para menor caminho)

Este algoritmo encontra o caminho mais curto de um ponto a outro em um grafo. Ele é eficaz quando todas as conexões (arestas) têm valores positivos, pois sempre escolhe o caminho mais curto que já foi encontrado e o utiliza para explorar novos caminhos.

Código exemplo:

```
import heapq


def dijkstra(grafo, inicio):
    """
    Encontra o caminho mais curto a partir do nó inicial.
    :param grafo: dicionário onde as chaves são nós e os valores são listas de tuplas (vizinhos, peso)
    :param inicio: nó de partida
    :return: dicionário com as distâncias mínimas a partir do nó inicial
    """
    distancias = {nó: float('inf') for nó in grafo}  # Inicializa distâncias como infinito
    distancias[inicio] = 0  # Distância do nó inicial para ele mesmo é 0
    fila = [(0, inicio)]  # Fila de prioridade para armazenar (peso, nó)

    while fila:
    peso_atual, nó_atual = heapq.heappop(fila)  # Pega o nó com menor peso
    for vizinho, peso in grafo[nó_atual]:
    nova_distancia = peso_atual + peso  # Calcula nova distância
    if nova_distancia < distancias[vizinho]:  # Se for menor, atualiza
    distancias[vizinho] = nova_distancia
    heapq.heappush(fila, (nova_distancia, vizinho))  # Adiciona à fila

    return distancias

# Testando Dijkstra
grafo = {
    'A': [('B', 1), ('C', 4)],
    'B': [('A', 1), ('C', 2), ('D', 5)],
```

```
    'C': [('A', 4), ('B', 2), ('D', 1)],
    'D': [('B', 5), ('C', 1)]
}
resultado_dijkstra = dijkstra(grafo, 'A')
print("Dijkstra:", resultado_dijkstra)
```

Algoritmo de Bellman-Ford (para menor caminho com pesos negativos)

O algoritmo Bellman-Ford é usado para encontrar o caminho mais curto em um grafo que pode ter arestas com pesos negativos. Ele verifica repetidamente todas as conexões para garantir que encontre o menor caminho, mesmo que algumas conexões tenham custos negativos.

Código exemplo:

```
def bellman_ford(grafo, inicio):
    """
    Encontra o caminho mais curto usando o algoritmo de Bellman-Ford.
    :param grafo: lista de arestas (nó1, nó2, peso)
    :param inicio: nó de partida
    :return: dicionário com as distâncias mínimas
    """
    distancias = {nó: float('inf') for nó in set(u for u, v, p in grafo) | set(v for u, v, p in grafo)}
    distancias[inicio] = 0

    for _ in range(len(distancias) - 1):
    for u, v, peso in grafo:
    if distancias[u] + peso < distancias[v]:  # Relaxamento da aresta
    distancias[v] = distancias[u] + peso
```

```
    return distancias


# Testando Bellman-Ford
grafo_bf = [
    ('A', 'B', 1),
    ('B', 'C', 2),
    ('A', 'C', 4),
    ('C', 'D', -3)
]
resultado_bellman = bellman_ford(grafo_bf, 'A')
print("Bellman-Ford:", resultado_bellman)
```

Algoritmo de Floyd-Warshall (para todos os pares mais curtos)

Este algoritmo calcula o menor caminho entre todos os pares de nós em um grafo. Ele considera todos os possíveis caminhos e atualiza as distâncias até que todas as possibilidades sejam examinadas, garantindo que todos os caminhos mais curtos sejam encontrados.

Código exemplo:

```
def floyd_warshall(grafo):
    """
    Encontra o menor caminho entre todos os pares de nós.
    :param grafo: dicionário de distâncias entre nós
    :return: matriz de distâncias
    """
    distancias = {nó: {v: float('inf') for v in grafo} for nó in grafo}
```

```
        for u in grafo:
        for v, peso in grafo[u].items():
        distancias[u][v] = peso
        distancias[u][u] = 0  # Distância de um nó para ele mesmo é 0

        for k in grafo:
        for i in grafo:
        for j in grafo:
        if distancias[i][j] > distancias[i][k] + distancias[k][j]:  # Relaxa as distâncias
                distancias[i][j] = distancias[i][k] + distancias[k][j]

        return distancias

# Testando Floyd-Warshall
grafo_fw = {
        'A': {'B': 1, 'C': 4},
        'B': {'C': 2},
        'C': {'D': 1},
        'D': {}
}
resultado_floyd = floyd_warshall(grafo_fw)
print("Floyd-Warshall:", resultado_floyd)
```

Algoritmo de Kruskal (para árvore geradora mínima)

O algoritmo de Kruskal é usado para construir uma árvore que conecta todos os nós de um grafo com o menor custo total, sem formar ciclos. Ele funciona selecionando as arestas mais leves uma a uma, garantindo que não haja conexões repetidas.

Código exemplo:

```
class DisjointSet:
        def __init__(self, n):
        self.pai = list(range(n))

        def find(self, u):
        if self.pai[u] != u:
        self.pai[u] = self.find(self.pai[u])  # Caminho comprimido
        return self.pai[u]

        def union(self, u, v):
        pai_u = self.find(u)
        pai_v = self.find(v)
        if pai_u != pai_v:
        self.pai[pai_v] = pai_u  # Une os conjuntos

def kruskal(nos, arestas):
        """
        Encontra a árvore geradora mínima usando Kruskal.
        :param nos: lista de nós
        :param arestas: lista de arestas (nó1, nó2, peso)
        :return: lista de arestas da árvore geradora mínima
        """
        arestas.sort(key=lambda x: x[2])  # Ordena arestas pelo peso
        ds = DisjointSet(len(nos))
        mst = []

        for u, v, peso in arestas:
```

```
            if ds.find(u) != ds.find(v):
                ds.union(u, v)
                mst.append((u, v, peso))

        return mst

# Testando Kruskal
nos_kruskal = [0, 1, 2, 3]
arestas_kruskal = [
    (0, 1, 1),
    (1, 2, 2),
    (0, 2, 4),
    (1, 3, 3),
    (2, 3, 1)
]
resultado_kruskal = kruskal(nos_kruskal, arestas_kruskal)
print("Kruskal:", resultado_kruskal)
```

Algoritmo de Prim (para árvore geradora mínima)

O algoritmo de Prim cria uma árvore geradora mínima começando de um nó específico e adicionando as arestas mais baratas que conectam nós ainda não visitados. Ele garante que todos os nós sejam conectados de forma eficiente.

Código exemplo:

```
import heapq
```

```
def prim(grafo, inicio):
    """
    Encontra a árvore geradora mínima usando Prim.
    :param grafo: dicionário onde as chaves são nós e os valores são listas de tuplas (vizinhos, peso)
    :param inicio: nó de partida
    :return: lista de arestas da árvore geradora mínima
    """
    mst = []
    visitados = set()
    fila = [(0, inicio, None)]  # (peso, nó, nó anterior)

    while fila:
    peso, nó_atual, nó_anterior = heapq.heappop(fila)
    if nó_atual not in visitados:
    visitados.add(nó_atual)
    if nó_anterior is not None:
    mst.append((nó_anterior, nó_atual, peso))  # Adiciona aresta à MST

    for vizinho, peso in grafo[nó_atual]:
    if vizinho not in visitados:
            heapq.heappush(fila, (peso, vizinho, nó_atual))  # Adiciona vizinhos à fila

    return mst

# Testando Prim
grafo_prim = {
    'A': [('B', 1), ('C', 4)],
```

```
    'B': [('A', 1), ('C', 2), ('D', 5)],
    'C': [('A', 4), ('B', 2), ('D', 1)],
    'D': [('B', 5), ('C', 1)]
}
resultado_prim = prim(grafo_prim, 'A')
print("Prim:", resultado_prim)
```

Busca em profundidade (DFS) e busca em largura (BFS) em grafo

DFS (Busca em Profundidade): Este método explora o grafo seguindo um caminho até o fim, antes de voltar e tentar outros caminhos. É como explorar um labirinto até chegar a uma parede e depois voltar.

BFS (Busca em Largura): Este método examina todos os vizinhos de um nó antes de descer para os próximos níveis. É como visitar todos os andares de um prédio antes de ir para o próximo.

Código exemplo:

```
# Implementação da Busca em Profundidade (DFS)
def dfs(grafo, nó, visitados=None):
    """
    Realiza a busca em profundidade em um grafo.
    :param grafo: dicionário onde as chaves são nós e os valores são listas de vizinhos
    :param nó: nó atual
    :param visitados: conjunto de nós já visitados
    """
    if visitados is None:
    visitados = set()  # Cria um conjunto para armazenar nós visitados
    visitados.add(nó)  # Marca o nó atual como visitado
    print(nó)  # Imprime o nó visitado
```

```
        for vizinho in grafo[nó]:  # Itera sobre os vizinhos do nó
        if vizinho not in visitados:  # Se o vizinho ainda não foi visitado
        dfs(grafo, vizinho, visitados)  # Chama a DFS recursivamente

# Implementação da Busca em Largura (BFS)
from collections import deque

def bfs(grafo, nó):
        """
        Realiza a busca em largura em um grafo.
        :param grafo: dicionário onde as chaves são nós e os valores são listas de vizinhos
        :param nó: nó inicial
        """
        visitados = set()  # Cria um conjunto para armazenar nós visitados
        fila = deque([nó])  # Fila para armazenar nós a serem visitados
        visitados.add(nó)  # Marca o nó inicial como visitado

        while fila:  # Enquanto houver nós na fila
        nó_atual = fila.popleft()  # Remove o nó da frente da fila
        print(nó_atual)  # Imprime o nó visitado

        for vizinho in grafo[nó_atual]:  # Itera sobre os vizinhos do nó atual
        if vizinho not in visitados:  # Se o vizinho ainda não foi visitado
        visitados.add(vizinho)  # Marca o vizinho como visitado
        fila.append(vizinho)  # Adiciona o vizinho à fila
```

```
# Testando DFS e BFS
grafo = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

print("Busca em Profundidade (DFS):")
dfs(grafo, 'A')  # Inicia a busca a partir do nó 'A'

print("\nBusca em Largura (BFS):")
bfs(grafo, 'A')  # Inicia a busca a partir do nó 'A'
```

Strings

Algoritmo de KMP (Knuth-Morris-Pratt)

O algoritmo de KMP é utilizado para encontrar substrings em uma string. Ele usa uma tabela de pré-processamento para evitar comparações desnecessárias, tornando a busca mais eficiente.

Código exemplo:

```
def kmp(pattern, text):
    # Função para construir a tabela de prefixos
    def build_lps(pattern):
    lps = [0] * len(pattern)
```

```
length = 0  # Comprimento do prefixo anterior
i = 1
while i < len(pattern):
if pattern[i] == pattern[length]:
length += 1
lps[i] = length
i += 1
else:
if length != 0:
        length = lps[length - 1]
else:
        lps[i] = 0
        i += 1
return lps

lps = build_lps(pattern)  # Cria a tabela de prefixos
i = j = 0  # Índices para o texto e o padrão
while i < len(text):
if pattern[j] == text[i]:
i += 1
j += 1

if j == len(pattern):  # Padrão encontrado
print(f'Padrão encontrado na posição {i - j}')
j = lps[j - 1]  # Continua a busca

elif i < len(text) and pattern[j] != text[i]:  # Não há correspondência
if j != 0:
```

```
            j = lps[j - 1]
        else:
            i += 1

# Testando o algoritmo de KMP
text = "ababcababcabc"
pattern = "abc"
kmp(pattern, text)
```

Algoritmo de Rabin-Karp (para busca de padrões em strings)

O algoritmo de Rabin-Karp é uma técnica de busca de padrões que utiliza a ideia de hashing. Ele calcula um valor hash para a substring de busca e compara esse hash com as substrings do texto, permitindo uma verificação rápida.

Código exemplo:

```
def rabin_karp(pattern, text, d=256, q=101):
    m = len(pattern)
    n = len(text)
    p = t = 0  # Hashes do padrão e do texto
    h = 1  # Valor hash para o primeiro bloco

    for i in range(m - 1):
        h = (h * d) % q  # Calcula h

    for i in range(m):
        p = (d * p + ord(pattern[i])) % q  # Hash do padrão
        t = (d * t + ord(text[i])) % q  # Hash do texto
```

```
        for i in range(n - m + 1):
        if p == t:  # Se os hashes são iguais
        if text[i:i + m] == pattern:  # Verifica a correspondência
        print(f'Padrão encontrado na posição {i}')

        if i < n - m:  # Calcula o hash para o próximo bloco
        t = (d * (t - ord(text[i]) * h) + ord(text[i + m])) % q
        t = (t + q) % q  # Corrige o valor negativo

# Testando o algoritmo de Rabin-Karp
text = "hello world"
pattern = "world"
rabin_karp(pattern, text)
```

Algoritmo de Manacher (para encontrar a maior substring palindrômica)

O algoritmo de Manacher é usado para encontrar a maior substring palindrômica em uma string. Ele utiliza uma abordagem eficiente que evita a necessidade de checar todos os substrings.

Código exemplo:

```
def manacher(s):
        # Adiciona separadores para lidar com palíndromos de tamanho par e ímpar
        T = '#'.join(f'^{s}$')
        n = len(T)
        P = [0] * n  # Array para armazenar o comprimento dos palíndromos
        C = R = 0  # Centro e limite do palíndromo mais longo encontrado
```

```
        for i in range(1, n - 1):
        mirror = 2 * C - i  # Cálculo do espelho

        if R > i:
        P[i] = min(R - i, P[mirror])  # Evita recalcular

        # Expande o palíndromo
        while T[i + P[i] + 1] == T[i - P[i] - 1]:
        P[i] += 1

        # Atualiza o centro e o limite
        if i + P[i] > R:
        C, R = i, i + P[i]

        # Encontra o comprimento do maior palíndromo
        max_length = max(P)
        center_index = P.index(max_length)
        start = (center_index - max_length) // 2  # Calcula o início da substring
original

        return s[start:start + max_length]

# Testando o algoritmo de Manacher
string = "babad"
print(f'A maior substring palindrômica é: "{manacher(string)}"')
```

Algoritmo de Z (para pesquisa de padrões)

O algoritmo de Z é usado para encontrar padrões em uma string. Ele cria um array Z que contém a extensão do prefixo mais longo da substring que é também um sufixo.

Código exemplo:

```
def z_algorithm(s):
    Z = [0] * len(s)
    L, R, K = 0, 0, 0

    for i in range(1, len(s)):
    if i > R:
    L, R = i, i
    while R < len(s) and s[R] == s[R - L]:
    R += 1
    Z[i] = R - L
    R -= 1
    else:
    K = i - L
    if Z[K] < R - i + 1:
    Z[i] = Z[K]
    else:
    L = i
    while R < len(s) and s[R] == s[R - L]:
        R += 1
    Z[i] = R - L
    R -= 1
```

```
    return Z

# Testando o algoritmo de Z
string = "abacab"
print(f'Array Z para "{string}": {z_algorithm(string)}')
```

Árvores

Árvores binárias de busca (Binary Search Trees)

Uma árvore binária de busca é uma estrutura de dados que possui a propriedade de que para cada nó, os valores dos nós da subárvore esquerda são menores e os valores dos nós da subárvore direita são maiores.

Código exemplo:

```
class Node:
        def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key


class BST:
        def __init__(self):
        self.root = None


        def insert(self, key):
        if self.root is None:
        self.root = Node(key)
        else:
        self._insert_rec(self.root, key)


        def _insert_rec(self, node, key):
        if key < node.val:
        if node.left is None:
        node.left = Node(key)
        else:
```

```
            self._insert_rec(node.left, key)
            else:
            if node.right is None:
            node.right = Node(key)
            else:
            self._insert_rec(node.right, key)

            def inorder(self):
            return self._inorder_rec(self.root)

            def _inorder_rec(self, node):
            return self._inorder_rec(node.left) + [node.val] +
self._inorder_rec(node.right) if node else []

# Testando a árvore binária de busca
bst = BST()
for key in [7, 3, 9, 1, 5, 8, 10]:
            bst.insert(key)

print(f'Travessia em ordem: {bst.inorder()}')  # Saída: [1, 3, 5, 7, 8, 9, 10]
```

Travessia em árvores (pré-ordem, pós-ordem, em ordem)

As travessias de uma árvore podem ser feitas em três ordens principais: pré-ordem, em ordem e pós-ordem.

Código exemplo:

```
class Traversal:
            def __init__(self, root):
```

```
        self.root = root

    def preorder(self):
        return self._preorder_rec(self.root)

    def _preorder_rec(self, node):
        return [node.val] + self._preorder_rec(node.left) + self._preorder_rec(node.right) if node else []

    def inorder(self):
        return self._inorder_rec(self.root)

    def _inorder_rec(self, node):
        return self._inorder_rec(node.left) + [node.val] + self._inorder_rec(node.right) if node else []

    def postorder(self):
        return self._postorder_rec(self.root)

    def _postorder_rec(self, node):
        return self._postorder_rec(node.left) + self._postorder_rec(node.right) + [node.val] if node else []

# Testando as travessias
traversal = Traversal(bst.root)
print(f'Travessia pré-ordem: {traversal.preorder()}')   # Saída: [7, 3, 1, 5, 9, 8, 10]
print(f'Travessia em ordem: {traversal.inorder()}')     # Saída: [1, 3, 5, 7, 8, 9, 10]
print(f'Travessia pós-ordem: {traversal.postorder()}')  # Saída: [1, 5, 3, 10, 8, 9, 7]
```

Árvores balanceadas (AVL, Árvores Rubro-Negras)

Uma árvore AVL é uma árvore binária de busca onde a altura de duas subárvores a partir de qualquer nó não difere em mais de uma unidade.

Código exemplo:

```
class AVLNode:
        def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key
        self.height = 1

class AVLTree:
        def insert(self, root, key):
        if not root:
        return AVLNode(key)
        elif key < root.val:
        root.left = self.insert(root.left, key)
        else:
        root.right = self.insert(root.right, key)

        root.height = 1 + max(self.get_height(root.left), self.get_height(root.right))
        balance = self.get_balance(root)

        # Rotação para a esquerda
        if balance > 1 and key < root.left.val:
        return self.rotate_right(root)
```

```
# Rotação para a direita
if balance < -1 and key > root.right.val:
return self.rotate_left(root)
# Rotação dupla (esquerda-direita)
if balance > 1 and key > root.left.val:
root.left = self.rotate_left(root.left)
return self.rotate_right(root)
# Rotação dupla (direita-esquerda)
if balance < -1 and key < root.right.val:
root.right = self.rotate_right(root.right)
return self.rotate_left(root)

return root

def rotate_left(self, z):
y = z.right
T2 = y.left
y.left = z
z.right = T2
z.height = 1 + max(self.get_height(z.left), self.get_height(z.right))
y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))
return y

def rotate_right(self, z):
y = z.left
T3 = y.right
y.right = z
z.left = T3
```

```
        z.height = 1 + max(self.get_height(z.left), self.get_height(z.right))
        y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))
        return y

        def get_height(self, root):
        return root.height if root else 0

        def get_balance(self, root):
        return self.get_height(root.left) - self.get_height(root.right)

        def inorder(self, root):
        return self._inorder_rec(root)

        def _inorder_rec(self, node):
        return self._inorder_rec(node.left) + [node.val] +
self._inorder_rec(node.right) if node else []

# Testando a árvore AVL
avl_tree = AVLTree()
root = None
for key in [10, 20, 30, 40, 50, 25]:
        root = avl_tree.insert(root, key)

print(f'Travessia em ordem da árvore AVL: {avl_tree.inorder(root)}')  # Saída: [10,
20, 25, 30, 40, 50]
```

Geometria Computacional

Algoritmo de Graham (Convex Hull)

O Algoritmo de Graham é usado para encontrar o convexo envoltório (Convex Hull) de um conjunto de pontos no plano

Código exemplo:

```
def orientation(p, q, r):
    """
    Retorna a orientação de três pontos:
    0 -> p, q e r são colineares
    1 -> Horário
    2 -> Antihorário
    """
    val = (q[1] - p[1]) * (r[0] - q[0]) - (q[0] - p[0]) * (r[1] - q[1])
    if val == 0:
    return 0
    return 1 if val > 0 else 2


def graham_scan(points):
    """
    Encontra o Convex Hull de um conjunto de pontos usando o Algoritmo de Graham.
    """
    # Encontra o ponto de referência (ponto mais baixo)
    points = sorted(points, key=lambda point: (point[1], point[0]))
    p0 = points[0]

    # Ordena os pontos em relação ao ponto de referência
```

```
    points = sorted(points, key=lambda point: (orientation(p0, point, (p0[0] + 1, p0[1])), point))

    # Cria a pilha para armazenar o Convex Hull
    hull = []

    for point in points:
    # Remove pontos que não fazem parte do Convex Hull
    while len(hull) > 1 and orientation(hull[-2], hull[-1], point) != 2:
    hull.pop()
    hull.append(point)

    return hull

# Testando o algoritmo de Graham
points = [(0, 3), (2, 2), (1, 1), (2, 1), (3, 0), (0, 0), (3, 3)]
hull = graham_scan(points)
print(f'O Convex Hull é: {hull}')  # Saída: [(0, 0), (3, 0), (3, 3), (0, 3)]
```

Teste de interseção de segmento de linha (Line Segment Intersection)

Para verificar se dois segmentos de linha se intersectam, podemos usar a orientação e o conceito de colinearidade.

Código exemplo:

```
def on_segment(p, q, r):
    """
    Verifica se o ponto q está no segmento de linha pr.
```

```
        """
        return (min(p[0], r[0]) <= q[0] <= max(p[0], r[0]) and
        min(p[1], r[1]) <= q[1] <= max(p[1], r[1]))

def do_intersect(p1, q1, p2, q2):
        """
        Verifica se os segmentos de linha p1q1 e p2q2 se intersectam.
        """
        o1 = orientation(p1, q1, p2)
        o2 = orientation(p1, q1, q2)
        o3 = orientation(p2, q2, p1)
        o4 = orientation(p2, q2, q1)

        # Casos gerais
        if o1 != o2 and o3 != o4:
        return True

        # Casos especiais
        if o1 == 0 and on_segment(p1, p2, q1): return True
        if o2 == 0 and on_segment(p1, q2, q1): return True
        if o3 == 0 and on_segment(p2, p1, q2): return True
        if o4 == 0 and on_segment(p2, q1, q2): return True

        return False

# Testando o teste de interseção
p1 = (1, 1)
q1 = (10, 1)
```

```
p2 = (1, 2)
q2 = (10, 2)

print(f'Os segmentos se intersectam? {do_intersect(p1, q1, p2, q2)}')  # Saída: False

p1 = (10, 0)
q1 = (0, 10)
p2 = (0, 0)
q2 = (10, 10)

print(f'Os segmentos se intersectam? {do_intersect(p1, q1, p2, q2)}')  # Saída: True
```

Algoritmo de varredura de linha (Line Sweep Algorithm)

O Algoritmo de Varredura de Linha é uma técnica utilizada em geometria computacional, frequentemente para resolver problemas de interseção de segmentos de linha. A implementação completa pode ser bastante longa, então aqui está uma versão simplificada que detecta interseções em um conjunto de segmentos.

Código exemplo:

```
class Event:
    def __init__(self, x, segment, is_start):
        self.x = x
        self.segment = segment
        self.is_start = is_start

def sweep_line(segments):
    """
```

```
    Detecta interseções entre segmentos de linha usando o Algoritmo de
Varredura de Linha.
    """
    events = []
    for segment in segments:
    events.append(Event(segment[0][0], segment, True))  # Ponto inicial
    events.append(Event(segment[1][0], segment, False))  # Ponto final

    # Ordena os eventos
    events.sort(key=lambda e: (e.x, not e.is_start))

    active_segments = []
    intersections = []

    for event in events:
    if event.is_start:
    # Adiciona o segmento ao conjunto ativo
    active_segments.append(event.segment)
    # Verifica interseções com segmentos ativos
    for active_segment in active_segments[:-1]:
    if do_intersect(active_segment[0], active_segment[1], event.segment[0],
event.segment[1]):
            intersections.append((active_segment, event.segment))
    else:
    # Remove o segmento do conjunto ativo
    active_segments.remove(event.segment)

    return intersections
```

```
# Testando o algoritmo de varredura de linha
segments = [((1, 1), (4, 4)), ((1, 4), (4, 1)), ((3, 2), (5, 2))]
intersections = sweep_line(segments)

print(f'Interseções encontradas: {intersections}')  # Saída: [(((1, 1), (4, 4)), ((1, 4), (4, 1)))]
```

Teoria dos Números

Algoritmo de Euclides (para encontrar o MDC)

O Algoritmo de Euclides é uma técnica eficiente para calcular o Máximo Divisor Comum (MDC) de dois números inteiros.

Código exemplo:

```
def euclidean_gcd(a, b):
    """
    Calcula o Máximo Divisor Comum (MDC) de dois números inteiros a e b usando o Algoritmo de Euclides.
    """
    while b:
    a, b = b, a % b  # Atualiza a e b
    return abs(a)  # Retorna o MDC absoluto

# Testando o Algoritmo de Euclides
num1 = 48
num2 = 18
print(f'O MDC de {num1} e {num2} é: {euclidean_gcd(num1, num2)}')  # Saída: 6
```

Teorema chinês do resto

O Teorema Chinês do Resto é uma maneira de resolver sistemas de congruências lineares. Ele afirma que, dado um conjunto de congruências, se os módulos são coprimos, existe uma solução única módulo do produto dos módulos.

Código exemplo:

```
def chinese_remainder_theorem(a, n):
    """
    Resolve o sistema de congruências usando o Teorema Chinês do Resto.
    a: lista de restos
    n: lista de módulos
    """
    from functools import reduce
    from math import gcd

    # Função para calcular o produto dos módulos
    def prod(lst):
    return reduce(lambda x, y: x * y, lst)

    # Verifica se os módulos são coprimos
    if len(a) != len(n):
    raise ValueError("As listas de restos e módulos devem ter o mesmo tamanho.")

    total = 0
    prod_n = prod(n)

    for ai, ni in zip(a, n):
    ni_product = prod_n // ni  # Produto dos módulos sem ni
```

```
        inv = pow(ni_product, -1, ni)  # Inverso modular
        total += ai * ni_product * inv

    return total % prod_n

# Testando o Teorema Chinês do Resto
a = [2, 3, 2]  # Restos
n = [3, 5, 7]  # Módulos

result = chinese_remainder_theorem(a, n)
print(f'A solução do sistema de congruências é: {result}')  # Saída: 23
```

Problema de Joséphus

O Problema de Joséphus é um enigma matemático clássico que envolve um grupo de soldados dispostos em um círculo. O processo de eliminação ocorre da seguinte forma: um número k é escolhido, e a cada k-ésima pessoa é removida até que reste apenas uma pessoa. O objetivo é determinar a posição da última pessoa sobrevivente.

Descrição do Problema

- Você tem nnn soldados, numerados de 0 a n−1.
- A cada k-ésima pessoa é eliminada.
- A questão é: qual a posição da última pessoa que sobrevive?

Recorrência

A solução pode ser expressa recursivamente:

Base: T(1,k)=0 (com apenas um soldado, a posição 0 sobrevive).

Recursão: T(n,k)=(T(n−1,k)+k) mod n

Aqui, a operação de módulo garante que as posições retornem a um índice válido, uma vez que após várias eliminações as posições dos soldados mudam.

Código exemplo:

```
def sobrevivente(n, k):
    """
    Função que encontra o sobrevivente no problema de Josephus.

    :param n: Número de soldados
    :param k: Passo de eliminação
    :return: A posição do sobrevivente (0-indexada)
    """
    # Caso base: se há apenas um soldado, ele sobrevive
    if n == 1:
    return 0  # Retorna 0, que é a posição do único soldado

    # Chama a função recursivamente para n-1 soldados
    # A fórmula (sobrevivente(n - 1, k) + k) % n calcula a nova posição
    return (sobrevivente(n - 1, k) + k) % n

def main():
    """
    Função principal para execução do programa.
    """
    # Lê o número de casos de teste
    NC = int(input("Digite o número de casos de teste: "))

    # Loop para processar cada caso de teste
```

```
    for i in range(1, NC + 1):
    # Lê n (número de soldados) e k (passo de eliminação)
    n, k = map(int, input(f"Digite n e k para o caso {i}: ").split())

    # Imprime o resultado para o caso i
    # Adiciona 1 para converter de posição 0-indexada para 1-indexada
    print(f"Case {i}: {sobrevivente(n, k) + 1}")

# Executa a função principal
if __name__ == "__main__":
    main()
```

Teoria dos números em geral (primes, divisores, etc.)

Aqui estão algumas funções básicas para trabalhar com números primos e divisores.

Código exemplo:

```
def is_prime(n):
    """
    Verifica se um número é primo.
    Um número é primo se for maior que 1 e não tiver divisores além de 1 e ele mesmo.
    """
    if n <= 1:
    return False
    for i in range(2, int(n**0.5) + 1):
    if n % i == 0:
    return False
```

```
        return True

def prime_factors(n):
        """
        Retorna a lista de fatores primos de um número n.
        """
        factors = []
        # Verifica os números 2 e maiores
        for i in range(2, int(n**0.5) + 1):
        while n % i == 0:
        factors.append(i)
        n //= i
        if n > 1:
        factors.append(n)  # Adiciona o último fator primo
        return factors

def divisors(n):
        """
        Retorna a lista de divisores de um número n.
        """
        divs = []
        for i in range(1, int(n**0.5) + 1):
        if n % i == 0:
        divs.append(i)
        if i != n // i:  # Adiciona o complemento do divisor
        divs.append(n // i)
        return sorted(divs)  # Retorna divisores ordenados
```

```
# Testando as funções de Teoria dos Números
num = 28
print(f'O número {num} é primo? {is_prime(num)}')  # Saída: False
print(f'Fatores primos de {num}: {prime_factors(num)}')  # Saída: [2, 2, 7]
print(f'Divisores de {num}: {divisors(num)}')  # Saída: [1, 2, 4, 7, 14, 28]
```

Tabela ASCII

Código:

```
# Função para exibir a tabela ASCII
def exibir_tabela_ascii():
    print("Tabela ASCII:")
    print("Código\tCaractere")
    for i in range(128):  # Os valores ASCII vão de 0 a 127
        print(f"{i}\t{chr(i)}")  # chr(i) converte o número em caractere

# Chamando a função
exibir_tabela_ascii()
```

Tabela

Caracter	Dec	Caracter2	Dec3	Caracter6	Dec7	Caracter10	Dec11
(nul)	0	@	64	Ç	128	+	192
(soh)	1	A	65	ü	129	-	193
(stx)	2	B	66	é	130	-	194
(etx)	3	C	67	â	131	+	195
(eot)	4	D	68	ä	132	-	196
(enq)	5	E	69	à	133	+	197
(ack)	6	F	70	å	134	ã	198
(bel)	7	G	71	ç	135	Ã	199
(bs)	8	H	72	ê	136	+	200
(ht)	9	I	73	ë	137	+	201

(nl)	10	J	74	è	138	-	202
(vt)	11	K	75	ï	139	-	203
(np)	12	L	76	î	140	¦	204
(cr)	13	M	77	ì	141	-	205
(so)	14	N	78	Ä	142	+	206
(si)	15	O	79	Å	143	¤	207
(dle)	16	P	80	É	144	ð	208
(dc1)	17	Q	81	æ	145	Ð	209
(dc2)	18	R	82	Æ	146	Ê	210
(dc3)	19	S	83	ô	147	Ë	211
(dc4)	20	T	84	ö	148	È	212
(nak)	21	U	85	ò	149	i	213
(syn)	22	V	86	û	150	Í	214
(etb)	23	W	87	ù	151	Î	215
(can)	24	X	88	ÿ	152	Ï	216
(em)	25	Y	89	Ö	153	+	217
(sub)	26	Z	90	Ü	154	+	218
(esc)	27	[	91	ø	155	_	219
(fs)	28	\	92	£	156	_	220
(gs)	29	]	93	Ø	157	¦	221
(rs)	30	^	94	×	1158	Ì	222
(us)	31	_	95	ƒ	159	_	223
(space)	32	`	96	á	160	Ó	224
!	33	a	97	í	161	ß	225
"	34	b	98	ó	162	Ô	226
#	35	c	99	ú	163	Ò	227
$	36	d	100	ñ	164	Õ	228
%	37	e	101	Ñ	165	Õ	229
&	38	f	102	ª	166	µ	230
'	39	g	103	º	167	Þ	231
(	40	h	104	¿	168	Þ	232
)	41	i	105	®	169	Ú	233
*	42	j	106	¬	170	Û	234
+	43	k	107	½	171	Ù	235
,	44	l	108	¼	172	ý	236
-	45	m	109	¡	173	Ý	237
.	46	n	110	«	174	¯	238
/	47	o	111	»	175	´	239

0	48	p	112	_	176		240
1	49	q	113	_	177	±	241
2	50	r	114	_	178	_	242
3	51	s	115	¦	179	¾	243
4	52	t	116	¦	180	¶	244
5	53	u	117	Á	181	§	245
6	54	v	118	Â	192	÷	24
7	55	w	119	À	183	¸	247
8	56	x	120	©	184	°	248
9	57	y	121	¦	185	¨	249
:	58	z	122	¦	186	·	250
;	59	{	123	+	187	¹	251
<	60	|	124	+	188	³	252
=	61	}	125	¢	189	²	253
>	62	~	126	¥	190	_	254
?	63	(del)	127	+	191		255