Entrada, cálculo e print com casas decimais

Programa completo que lê dois valores, calcula resultados e controla exatamente a apresentação decimal.

Entrada: dois números. Saída: soma, média e produto com 2 casas.

Tempo O(1) · memória O(1)

EXEMPLO DE CÓDIGO

solucao.py

```python
import sys


def main():
    # split separa a linha; map converte cada parte para float.
    a, b = map(float, input().split())

    # Os cálculos usam a precisão original dos valores.
    soma = a + b
    media = soma / 2.0
    produto = a * b

    # :.2f arredonda apenas na apresentação e sempre mostra 2 casas.
    print(f"Soma = {soma:.2f}")
    print(f"Media = {media:.2f}")
    print(f"Produto = {produto:.2f}")

    # O juiz normalmente espera uma quebra após a última linha.
    # print já acrescenta essa quebra automaticamente.


if __name__ == "__main__":
    main()
```

TÉCNICA

input() devolve texto; a conversão define quais operações podem ser feitas. A f-string separa cálculo de formatação.

ARMADILHA

Não arredonde antes de terminar os cálculos. Confira maiúsculas, rótulos, espaços e número de casas exigidos.

T casos com if, elif e else

Programa completo que processa uma quantidade declarada de notas e classifica cada caso por intervalos.

Entrada: T e uma nota por caso. Saída: classificação por linha.

Tempo O(T) · memória O(1)

EXEMPLO DE CÓDIGO

solucao.py

```python
def classificar(nota):
    # A ordem impede que um intervalo capture outro.
    if 90 <= nota <= 100:
        return "A"
    elif 70 <= nota < 90:
        return "B"
    elif 60 <= nota < 70:
        return "C"
    elif 0 <= nota < 60:
        return "D"
    else:
        return "nota invalida"


def main():
    # A primeira linha informa quantos casos virão.
    quantidade = int(input())

    for caso in range(1, quantidade + 1):
        nota = int(input())
        conceito = classificar(nota)
        print(f"Caso {caso}: {conceito}")


if __name__ == "__main__":
    main()
```

TÉCNICA

for range(1, T + 1) produz casos numerados de 1 a T. A função devolve a decisão; o main cuida do protocolo.

ARMADILHA

Teste 0, 59, 60, 69, 70, 89, 90 e 100. Bordas inclusivas são fonte frequente de WA.

while, sentinela, break e continue

Programa completo que lê valores até zero, ignora negativos e acumula somente os positivos válidos.

Entrada: inteiros até aparecer 0. Saída: quantidade e soma.

Tempo O(n) · memória O(1)

EXEMPLO DE CÓDIGO

solucao.py

```python
def main():
    quantidade = 0
    soma = 0

    while True:
        valor = int(input())

        # Zero encerra a entrada e não é processado.
        if valor == 0:
            break

        # Negativos são ignorados nesta regra.
        if valor < 0:
            continue

        # Só valores positivos chegam até aqui.
        quantidade += 1
        soma += valor

    print(f"Quantidade: {quantidade}")
    print(f"Soma: {soma}")


if __name__ == "__main__":
    main()
```

TÉCNICA

break encerra o laço inteiro. continue pula apenas o restante da repetição atual e volta à leitura.

ARMADILHA

Em laços com índice, atualize o estado antes de um possível continue; caso contrário o mesmo estado pode repetir para sempre.

Leitura até EOF preservando a linha

Programa completo que recebe frases até o fim do arquivo e conta letras, dígitos e espaços em cada uma.

Entrada: número desconhecido de linhas. Saída: três contagens ...

Tempo O(total de caracteres) · memória O(1) extra

EXEMPLO DE CÓDIGO

solucao.py

```python
import sys


def analisar(texto):
    letras = 0
    digitos = 0
    espacos = 0

    for caractere in texto:
        if caractere.isalpha():
            letras += 1
        elif caractere.isdigit():
            digitos += 1
        elif caractere == " ":
            espacos += 1

    return letras, digitos, espacos


def main():
    # O for termina naturalmente quando o arquivo acaba.
    for linha in sys.stdin:
        # Remove apenas a quebra, preservando outros espaços.
        texto = linha.rstrip("\n")
        l, d, e = analisar(texto)
        print(l, d, e)


if __name__ == "__main__":
    main()
```

TÉCNICA

EOF não é um valor da entrada: é a ausência de outra linha. for linha in sys.stdin expressa esse formato diretamente.

ARMADILHA

strip() também remove espaços significativos das pontas. Use rstrip('\n') quando o conteúdo completo da linha importa.

Entrada e saída rápidas em bloco

Programa completo para muitos tokens: lê tudo como bytes, consome com iterador e escreve todas as respostas de uma vez.

Entrada: T listas numéricas. Saída: soma de cada lista.

Tempo O(total de tokens) · memória O(total da entrada)

EXEMPLO DE CÓDIGO

solucao.py

```python
import sys


def main():
    # read().split() é rápido quando espaços e linhas são equivalentes.
    dados = iter(sys.stdin.buffer.read().split())
    quantidade_de_casos = int(next(dados))
    respostas = []

    for _ in range(quantidade_de_casos):
        tamanho = int(next(dados))
        soma = 0

        for _ in range(tamanho):
            soma += int(next(dados))

        # Guardamos texto para uma única escrita no final.
        respostas.append(str(soma))

    sys.stdout.write("\n".join(respostas) + "\n")


if __name__ == "__main__":
    main()
```

TÉCNICA

O iterador consome exatamente os tokens do caso atual. join reduz milhares de chamadas de escrita.

ARMADILHA

Essa técnica perde a distinção entre linhas. Não a use quando frases, linhas vazias ou espaços internos fazem parte do dado.

for, range, enumerate e matriz

Programa completo que lê uma matriz, percorre valores e índices e encontra a posição do maior elemento.

Entrada: linhas, colunas e matriz. Saída: soma e posição do má...

Tempo O(linhas·colunas) · memória O(linhas·colunas)

EXEMPLO DE CÓDIGO

solucao.py

```python
def main():
    linhas, colunas = map(int, input().split())
    matriz = [
        list(map(int, input().split()))
        for _ in range(linhas)
    ]

    soma = 0
    maior = matriz[0][0]
    melhor_linha = 0
    melhor_coluna = 0

    for i, linha in enumerate(matriz):
        for j, valor in enumerate(linha):
            soma += valor

            if valor > maior:
                maior = valor
                melhor_linha = i
                melhor_coluna = j

    print(f"Soma: {soma}")
    print(f"Maior: {maior}")
    print(f"Posicao: {melhor_linha + 1} {melhor_coluna + 1}")


if __name__ == "__main__":
    main()
```

TÉCNICA

enumerate entrega posição e valor. Índices internos começam em 0; a saída pode exigir conversão para base 1.

ARMADILHA

Não assuma matriz não vazia sem conferir as restrições. Cada linha também precisa conter a quantidade declarada de colunas.

Varredura de lista com melhor resposta

Programa completo que encontra o palpite mais próximo e preserva o primeiro participante quando há empate.

Entrada: T casos, alvo e palpites. Saída: posição vencedora.

Tempo O(total de palpites) · memória O(n)

EXEMPLO DE CÓDIGO

solucao.py

```python
N = int(input().strip())

for _ in range(N):
    QT, S = [int(x) for x in input().strip().split(' ')]

    palpites = [int(x) for x in input().strip().split(' ')]

    # Começamos considerando o primeiro aluno como o melhor.
    melhor_palpite = 0
    menor_diferenca = abs(S - palpites[0])

    # Só trocamos em uma diferença estritamente menor; assim, empates mantêm
    # o primeiro palpite, como exige o enunciado.
    for i in range(1, QT):
        if(menor_diferenca > abs(S - palpites[i])):
            melhor_palpite = i
            menor_diferenca = abs(S - palpites[i])

    print(melhor_palpite + 1)
```

TÉCNICA

A melhor resposta começa no primeiro elemento. A comparação estrita atualiza apenas quando a nova distância é realmente menor.

ARMADILHA

Trocar <= por < altera o desempate. A posição impressa é índice + 1.

Funções, math e estatística básica

Programa completo que decompõe cálculos em funções e imprime distância, média e mediana.

Entrada: dois pontos e uma lista. Saída: medidas com 3 casas.

Tempo O(n log n) pela ordenação · memória O(n)

EXEMPLO DE CÓDIGO

solucao.py

```python
import math


def distancia(x1, y1, x2, y2):
    # hypot calcula sqrt(dx² + dy²).
    return math.hypot(x1 - x2, y1 - y2)


def media(valores):
    return sum(valores) / len(valores)


def mediana(valores):
    ordenados = sorted(valores)
    n = len(ordenados)
    meio = n // 2

    if n % 2 == 1:
        return ordenados[meio]
    return (ordenados[meio - 1] + ordenados[meio]) / 2


def main():
    x1, y1, x2, y2 = map(float, input().split())
    n = int(input())
    valores = list(map(float, input().split()))
    assert len(valores) == n

    print(f"Distancia: {distancia(x1, y1, x2, y2):.3f}")
    print(f"Media: {media(valores):.3f}")
    print(f"Mediana: {mediana(valores):.3f}")


if __name__ == "__main__":
    main()
```

TÉCNICA

Funções separam contratos pequenos: parâmetros entram, return devolve o resultado e print fica no programa principal.

ARMADILHA

math.sin recebe radianos. Quando o enunciado fornece graus, converta com math.radians(angulo).

Dicionário e set para frequências

Programa completo que conta palavras, remove duplicatas e responde consultas de ocorrência.

Entrada: palavras e consultas. Saída: frequência ou zero.

Tempo O(n+q) médio · memória O(n)

EXEMPLO DE CÓDIGO

solucao.py

```python
def main():
    n, q = map(int, input().split())
    palavras = input().split()
    assert len(palavras) == n

    frequencia = {}
    unicas = set()

    for palavra in palavras:
        frequencia[palavra] = frequencia.get(palavra, 0) + 1
        unicas.add(palavra)

    print(f"Distintas: {len(unicas)}")

    for _ in range(q):
        consulta = input().strip()
        print(frequencia.get(consulta, 0))


if __name__ == "__main__":
    main()
```

TÉCNICA

O dicionário associa palavra à contagem. O set responde pertencimento e quantidade de valores distintos.

ARMADILHA

set não preserva duplicatas. A ordem de iteração não deve ser usada como regra de saída, salvo garantia explícita.

Ordenação com múltiplos critérios

Programa completo que ordena registros por peso decrescente, idade, altura e nome crescentes.

Entrada: cenários de renas. Saída: prefixo do ranking.

Tempo O(n log n) · memória O(n)

EXEMPLO DE CÓDIGO

solucao.py

```python
quantidade_de_cenarios = int(input())

for numero_do_cenario in range(1, quantidade_de_cenarios + 1):
    quantidade_de_renas, quantidade_escolhida = map(int, input().split())
    renas = []

    for _ in range(quantidade_de_renas):
        nome, peso, idade, altura = input().split()

        # Cada tupla funciona como um registro: os quatro campos da rena
        # permanecem juntos durante toda a ordenação.
        renas.append((nome, int(peso), int(idade), float(altura)))

    # As tuplas-chave são comparadas campo a campo:
    # peso decrescente, idade crescente, altura crescente e nome crescente.
    renas.sort(
        key=lambda rena: (
            -rena[1],
            rena[2],
            rena[3],
            rena[0],
        )
    )

    print(f"CENARIO {{{numero_do_cenario}}}")

    # A lista já está no ranking correto; basta imprimir o prefixo solicitado.
    for posicao, rena in enumerate(renas[:quantidade_escolhida], start=1):
        print(f"{posicao} - {rena[0]}")
```

TÉCNICA

A chave é uma tupla comparada em cascata. O sinal negativo transforma apenas o peso em ordem decrescente.

ARMADILHA

reverse=True inverteria todos os critérios. A tupla permite misturar sentidos e desempates corretamente.

Busca binária da primeira ocorrência

Programa completo que ordena mármores uma vez e usa bisect_left em cada consulta.

Entrada: vários casos com valores e consultas. Saída: primeira...

O(n log n + q log n) por caso

EXEMPLO DE CÓDIGO

solucao.py

```python
# bisect_left faz busca binária e devolve a primeira posição possível.
from bisect import bisect_left


# O contador produz o cabeçalho CASE# pedido pelo enunciado.
caso = 0

while True:
    n, q = map(int, input().split())

    # O par 0 0 encerra a entrada.
    if n == 0 and q == 0:
        break

    caso += 1

    valores = []

    for _ in range(n):
        valor = int(input())
        valores.append(valor)

    # A ordenação é o pré-processamento necessário para todas as consultas.
    valores.sort()

    print(f"CASE# {caso}:")

    for _ in range(q):
        valor = int(input())

        # Procuramos a primeira ocorrência, não apenas qualquer ocorrência.
        pos = bisect_left(valores, valor)

        # O ponto de inserção ainda precisa ser validado.
        if pos < len(valores) and valores[pos] == valor:
            print(f"{valor} found at {pos + 1}")
        else:
            print(f"{valor} not found")
```

TÉCNICA

bisect_left devolve o ponto mais à esquerda onde o alvo poderia entrar. O código ainda confirma se ele existe.

ARMADILHA

A busca depende da ordenação. A posição interna começa em 0, mas o enunciado imprime começando em 1.

Pilha: balanço de parênteses até EOF

Programa completo do beecrowd 1068. A pilha representa aberturas que ainda aguardam fechamento.

Entrada: expressões até EOF. Saída: correct ou incorrect.

Tempo O(n) · memória O(n) por expressão

EXEMPLO DE CÓDIGO

solucao.py

```python
import sys


def esta_balanceada(expressao):
    """Verifica se os parênteses de uma expressão abrem e fecham na ordem correta."""
    pilha = []

    for caractere in expressao:
        if caractere == "(":
            # A abertura fica pendente até aparecer o fechamento correspondente.
            pilha.append(caractere)
        elif caractere == ")":
            # Fechar sem nenhuma abertura disponível invalida a expressão.
            if not pilha:
                return False
            pilha.pop()

    # Se ainda há itens na pilha, sobraram aberturas sem fechamento.
    return not pilha


# Cada linha da entrada é uma expressão; a leitura termina no fim do arquivo.
for linha in sys.stdin:
    expressao = linha.rstrip("\n")
    print("correct" if esta_balanceada(expressao) else "incorrect")
```

TÉCNICA

Ao ler '(', empilhe. Ao ler ')', precisa existir uma abertura disponível. No final, a pilha deve estar vazia.

ARMADILHA

Contagens iguais não garantem ordem correta. ')(' deve ser rejeitada imediatamente.

Fila com deque: jogando cartas fora

Programa completo do beecrowd 1110. O topo fica no front e a base no back do deque.

Entrada: valores n até sentinela 0. Saída: descartes e restante.

Tempo O(n) · memória O(n) por caso

EXEMPLO DE CÓDIGO

solucao.py

```python
from collections import deque


while True:
    quantidade = int(input())

    # O valor zero não representa um baralho: ele encerra a entrada.
    if quantidade == 0:
        break

    # O início do deque é o topo do baralho e o fim é a base.
    cartas = deque(range(1, quantidade + 1))
    descartadas = []

    while len(cartas) > 1:
        # A carta do topo sai definitivamente do baralho.
        descartadas.append(cartas.popleft())

        # A nova carta do topo é retirada e recolocada na base.
        cartas.append(cartas.popleft())

    texto_descartadas = ", ".join(map(str, descartadas))
    print(f"Discarded cards: {texto_descartadas}")
    print(f"Remaining card: {cartas[0]}")
```

TÉCNICA

popleft descarta o topo; append(popleft()) move o novo topo para a base. Cada operação nas pontas custa O(1).

ARMADILHA

list.pop(0) desloca os demais elementos e custa O(n). Para fila, prefira collections.deque.

Pilha, fila e heap simulados em paralelo

Programa completo do beecrowd 1340. Cada remoção elimina as estruturas que não poderiam devolver o valor observado.

Entrada: casos até EOF com operações 1 x e 2 x. Saída: classif...

Tempo O(n log n) · memória O(n)

EXEMPLO DE CÓDIGO

solucao.py

```python
from collections import deque
import heapq
import sys


def classificar(operacoes):
    """Simula as três estruturas e classifica quais reproduzem as remoções."""
    pilha = []
    fila = deque()
    prioridade = []  # heapq é min-heap; valores negativos simulam uma max-heap.

    pode_ser_pilha = True
    pode_ser_fila = True
    pode_ser_prioridade = True

    for tipo, valor in operacoes:
        if tipo == 1:
            # Toda inserção é reproduzida nas candidatas que ainda são possíveis.
            if pode_ser_pilha:
                pilha.append(valor)
            if pode_ser_fila:
                fila.append(valor)
            if pode_ser_prioridade:
                heapq.heappush(prioridade, -valor)
            continue

        # Na remoção, o valor observado precisa ser exatamente o que a estrutura
        # candidata retiraria naquele momento.
        if pode_ser_pilha and (not pilha or pilha.pop() != valor):
            pode_ser_pilha = False
        if pode_ser_fila and (not fila or fila.popleft() != valor):
            pode_ser_fila = False
        if pode_ser_prioridade and (
            not prioridade or -heapq.heappop(prioridade) != valor
        ):
            pode_ser_prioridade = False

    candidatas = sum((pode_ser_pilha, pode_ser_fila, pode_ser_prioridade))
    if candidatas == 0:
        return "impossible"
    if candidatas > 1:
        return "not sure"
    if pode_ser_pilha:
        return "stack"
    if pode_ser_fila:
        return "queue"
    return "priority queue"


# Os casos aparecem até EOF. Usar um iterador facilita consumir exatamente
# a quantidade de operações declarada em cada caso.
dados = iter(map(int, sys.stdin.buffer.read().split()))

while True:
    try:
        quantidade = next(dados)
    except StopIteration:
        break

    operacoes = [(next(dados), next(dados)) for _ in range(quantidade)]
    print(classificar(operacoes))
```

TÉCNICA

A mesma inserção é reproduzida nas três candidatas. Uma remoção compara LIFO, FIFO e maior prioridade.

ARMADILHA

heapq é min-heap. Valores negativos simulam a retirada do maior elemento.

Transformação de string em três fases

Programa completo do beecrowd 1024. Cada regra recebe o resultado da fase anterior.

Entrada: quantidade e mensagens completas. Saída: mensagem cri...

Tempo O(n) · memória O(n) por mensagem

EXEMPLO DE CÓDIGO

solucao.py

```python
import sys


def criptografar(mensagem):
    """Executa, na ordem, as três fases descritas pelo enunciado."""
    # Fase 1: somente letras ASCII avançam três códigos.
    deslocada = [
        chr(ord(caractere) + 3)
        if "A" <= caractere <= "Z" or "a" <= caractere <= "z"
        else caractere
        for caractere in mensagem
    ]
    # Fase 2: a mensagem inteira é invertida.
    invertida = deslocada[::-1]
    metade = len(invertida) // 2
    # Fase 3: os caracteres da metade final recuam um código.
    return "".join(
        invertida[:metade]
        + [chr(ord(caractere) - 1) for caractere in invertida[metade:]]
    )


def main():
    # A primeira linha informa quantas mensagens devem ser transformadas.
    linhas = sys.stdin.buffer.read().decode().splitlines()
    quantidade = int(linhas[0])
    respostas = [criptografar(linhas[i]) for i in range(1, quantidade + 1)]
    # write não acrescenta quebra automaticamente; fechamos também a última linha.
    sys.stdout.write("\n".join(respostas) + "\n")


if __name__ == "__main__":
    main()
```

TÉCNICA

ord e chr deslocam códigos; [::-1] inverte; len // 2 define a metade após a inversão.

ARMADILHA

Executar fases fora de ordem muda as posições usadas pela terceira regra.

Máquina de estados em uma frase

Programa completo do beecrowd 1234. O booleano informa como a próxima letra deve ser escrita.

Entrada: sentenças até EOF. Saída: maiúsculas e minúsculas alt...

Tempo O(n) · memória O(n) por linha

EXEMPLO DE CÓDIGO

solucao.py

```python
import sys


def transformar_sentenca(sentenca):
    """Alterna a caixa das letras sem deixar espaços alterarem o estado."""
    usar_maiuscula = True
    resposta = []

    for caractere in sentenca:
        if caractere.isalpha():
            # O estado determina a caixa desta letra e muda para a próxima.
            resposta.append(
                caractere.upper() if usar_maiuscula else caractere.lower()
            )
            usar_maiuscula = not usar_maiuscula
        else:
            # Espaços e sinais são copiados sem consumir a alternância.
            resposta.append(caractere)

    return "".join(resposta)


def main():
    # O problema fornece sentenças até o fim do arquivo.
    linhas = sys.stdin.buffer.read().decode().splitlines()
    sys.stdout.write("\n".join(map(transformar_sentenca, linhas)) + "\n")


if __name__ == "__main__":
    main()
```

TÉCNICA

O estado troca somente depois de uma letra. Espaços e sinais são copiados sem consumir a alternância.

ARMADILHA

Usar o índice absoluto falha quando espaços não devem contar.

Intercalação com sobra da maior string

Programa completo do beecrowd 1238. Intercala o prefixo comum e anexa o restante sem perder caracteres.

Entrada: T pares de strings. Saída: uma combinação por linha.

Tempo O(|a|+|b|) · memória O(|a|+|b|)

EXEMPLO DE CÓDIGO

solucao.py

```python
import sys


def combinar(a, b):
    """Intercala o prefixo comum e acrescenta a sobra da string maior."""
    # Intercalamos enquanto as duas strings possuem a posição i.
    limite = min(len(a), len(b))
    intercalada = "".join(
        a[i] + b[i] for i in range(limite)
    )

    # Apenas uma das duas sobras será não vazia.
    return intercalada + a[limite:] + b[limite:]


def main():
    # A primeira linha informa quantos pares devem ser combinados.
    linhas = sys.stdin.buffer.read().decode().splitlines()
    quantidade = int(linhas[0])
    respostas = []

    for i in range(1, quantidade + 1):
        a, b = linhas[i].split()
        respostas.append(combinar(a, b))

    # Uma única escrita costuma ser mais eficiente do que vários prints.
    sys.stdout.write("\n".join(respostas) + "\n")


if __name__ == "__main__":
    main()
```

TÉCNICA

min(len(a), len(b)) define quantos pares existem. Depois do limite, somente uma das sobras é não vazia.

ARMADILHA

zip(a, b) puro descarta a sobra da string maior. Ela precisa ser anexada explicitamente.

DP da maior substring comum

Programa completo do beecrowd 1237. Coincidências estendem a diagonal anterior; divergências valem zero.

Entrada: pares de linhas até EOF. Saída: tamanho do maior trec...

Tempo O(n·m) · memória O(min(n,m))

EXEMPLO DE CÓDIGO

solucao.py

```python
import sys


def maior_substring_comum(a, b):
    """Calcula a maior substring comum com DP de memória otimizada."""
    # b será a menor string para reduzir a memória para O(min(n, m)).
    if len(b) > len(a):
        a, b = b, a

    # anterior representa a linha anterior da tabela de programação dinâmica.
    anterior = [0] * (len(b) + 1)
    melhor = 0

    for caractere_a in a:
        atual = [0] * (len(b) + 1)
        for j, caractere_b in enumerate(b, start=1):
            if caractere_a == caractere_b:
                # A coincidência estende a diagonal; diferenças permanecem zero
                # porque uma substring não pode pular caracteres.
                atual[j] = anterior[j - 1] + 1
                melhor = max(melhor, atual[j])
        anterior = atual

    return melhor


def main():
    # Cada caso possui duas linhas e a entrada termina em EOF.
    linhas = sys.stdin.buffer.read().decode().splitlines()
    respostas = []
    for i in range(0, len(linhas) - 1, 2):
        respostas.append(str(maior_substring_comum(linhas[i], linhas[i + 1])))
    sys.stdout.write("\n".join(respostas) + "\n")


if __name__ == "__main__":
    main()
```

TÉCNICA

anterior[j-1] representa o sufixo comum que terminava nas duas posições anteriores.

ARMADILHA

Substring é contínua; subsequência permite saltos e resolve outro problema.

DFS e árvore de descoberta

Programa completo do beecrowd 1076. A DFS conta arestas que alcançam vértices novos e dobra para ida e volta.

Entrada: T labirintos. Saída: movimentos mínimos de caneta.

Tempo O(V+E) · memória O(V+E)

EXEMPLO DE CÓDIGO

solucao.py

```python
def dfs(vertice, grafo, visitado):
    """Visita o componente e devolve quantas arestas descobriram vértices novos."""
    visitado[vertice] = True
    arestas_da_busca = 0

    # A DFS segue cada vizinho ainda não visitado e aprofunda a busca.
    for vizinho in grafo[vertice]:
        if not visitado[vizinho]:
            # Esta aresta entra na árvore da DFS porque alcançou um vértice novo.
            arestas_da_busca += 1
            arestas_da_busca += dfs(vizinho, grafo, visitado)

    return arestas_da_busca


# O primeiro valor informa quantos labirintos serão processados.
quantidade_de_casos = int(input())

for _ in range(quantidade_de_casos):
    origem = int(input())
    vertices, arestas = map(int, input().split())

    # Cada posição guarda somente os vizinhos do respectivo vértice.
    grafo = [[] for _ in range(vertices)]

    for _ in range(arestas):
        a, b = map(int, input().split())
        # O labirinto é um grafo não direcionado, então registramos os dois sentidos.
        grafo[a].append(b)
        grafo[b].append(a)

    visitado = [False] * vertices
    arestas_da_busca = dfs(origem, grafo, visitado)

    # Para sair da origem e voltar a ela, cada aresta da árvore é percorrida
    # uma vez na ida e outra na volta.
    print(arestas_da_busca * 2)
```

TÉCNICA

A lista de adjacência guarda vizinhos. visitado evita repetir vértices e neutraliza arestas duplicadas.

ARMADILHA

Em grafo não direcionado, registre a e b nos dois sentidos. Em direcionado, não invente a volta.

BFS no tabuleiro do cavalo

Programa completo do beecrowd 1100. A fila visita casas em camadas de distância crescente.

Entrada: pares de casas até EOF. Saída: menor número de movime...

O(64·8) por caso · memória O(64)

EXEMPLO DE CÓDIGO

solucao.py

```python
from collections import deque
import sys


# Os oito deslocamentos possíveis de um cavalo: duas casas em um eixo
# e uma casa no outro eixo.
MOVIMENTOS = (
    (2, 1), (2, -1), (-2, 1), (-2, -1),
    (1, 2), (1, -2), (-1, 2), (-1, -2),
)


def para_coordenada(posicao):
    """Converte uma casa como 'a1' para índices de linha e coluna entre 0 e 7."""
    coluna = ord(posicao[0]) - ord("a")
    linha = int(posicao[1]) - 1
    return linha, coluna


def esta_no_tabuleiro(linha, coluna):
    """Informa se a coordenada pertence ao tabuleiro 8 x 8."""
    return 0 <= linha < 8 and 0 <= coluna < 8


def bfs(origem, destino):
    """Calcula o menor número de movimentos entre duas casas com BFS."""
    inicio = para_coordenada(origem)
    fim = para_coordenada(destino)

    # A fila mantém as casas por camadas de distância.
    fila = deque([(inicio[0], inicio[1], 0)])
    visitado = [[False] * 8 for _ in range(8)]
    visitado[inicio[0]][inicio[1]] = True

    while fila:
        linha, coluna, distancia = fila.popleft()

        # A primeira retirada do destino ocorre pelo menor caminho possível.
        if (linha, coluna) == fim:
            return distancia

        for delta_linha, delta_coluna in MOVIMENTOS:
            nova_linha = linha + delta_linha
            nova_coluna = coluna + delta_coluna

            if (
                esta_no_tabuleiro(nova_linha, nova_coluna)
                and not visitado[nova_linha][nova_coluna]
            ):
                # Marcar ao entrar na fila impede que a mesma casa seja enfileirada
                # várias vezes por caminhos diferentes.
                visitado[nova_linha][nova_coluna] = True
                fila.append((nova_linha, nova_coluna, distancia + 1))

    return -1  # Não ocorre em um tabuleiro comum, mas completa o contrato da função.


# O problema fornece pares de casas até o fim do arquivo.
for linha_de_entrada in sys.stdin:
    linha_de_entrada = linha_de_entrada.strip()
    if not linha_de_entrada:
        continue

    origem, destino = linha_de_entrada.split()
    resposta = bfs(origem, destino)
    print(f"To get from {origem} to {destino} takes {resposta} knight moves.")
```

TÉCNICA

Os oito deslocamentos formam as arestas do grafo. A primeira retirada do destino tem a menor distância.

ARMADILHA

Marque a casa quando ela entra na fila, não quando sai, para evitar inserções duplicadas.

Dijkstra com lista e fila de prioridade

Programa completo para menor caminho com pesos não negativos, usando heap e descarte de entradas antigas.

Entrada: grafo direcionado e uma origem. Saída: distâncias ou -1.

O((V+E) log V) · memória O(V+E)

EXEMPLO DE CÓDIGO

solucao.py

```python
import heapq
import sys


def dijkstra(grafo, origem):
    infinito = 10**18
    distancia = [infinito] * len(grafo)
    distancia[origem] = 0
    fila = [(0, origem)]

    while fila:
        custo_atual, vertice = heapq.heappop(fila)

        # Uma rota melhor já foi inserida depois desta.
        if custo_atual != distancia[vertice]:
            continue

        for vizinho, peso in grafo[vertice]:
            novo_custo = custo_atual + peso

            # Relaxamento: melhora a estimativa do vizinho.
            if novo_custo < distancia[vizinho]:
                distancia[vizinho] = novo_custo
                heapq.heappush(fila, (novo_custo, vizinho))

    return distancia


def main():
    vertices, arestas, origem = map(int, input().split())
    grafo = [[] for _ in range(vertices)]

    for _ in range(arestas):
        a, b, peso = map(int, input().split())
        grafo[a].append((b, peso))

    resposta = dijkstra(grafo, origem)
    print(*(-1 if d == 10**18 else d for d in resposta))


if __name__ == "__main__":
    main()
```

TÉCNICA

Relaxar é testar se passar pelo vértice atual cria custo menor. A heap sempre oferece a menor estimativa.

ARMADILHA

Dijkstra não funciona com pesos negativos. BFS minimiza quantidade de arestas, não a soma dos pesos.

Círculos: contenção e separação

Programa completo com dois tipos de consulta: círculo dentro de outro e dois círculos em um retângulo.

Entrada: T consultas DENTRO ou RETANGULO. Saída: S ou N.

Tempo O(T) · memória O(1)

EXEMPLO DE CÓDIGO

solucao.py

```python
def circulo_dentro(r1, x1, y1, r2, x2, y2):
    # Um círculo maior nunca cabe no menor.
    if r2 > r1:
        return False

    dx = x1 - x2
    dy = y1 - y2
    limite = r1 - r2

    # Distância entre centros <= diferença dos raios.
    return dx * dx + dy * dy <= limite * limite


def cabem_no_retangulo(largura, altura, r1, r2):
    # Cada diâmetro precisa caber isoladamente.
    for raio in (r1, r2):
        if 2 * raio > largura or 2 * raio > altura:
            return False

    # Cantos opostos maximizam a distância.
    dx = largura - r1 - r2
    dy = altura - r1 - r2
    return dx * dx + dy * dy >= (r1 + r2) ** 2


def main():
    t = int(input())

    for _ in range(t):
        partes = input().split()
        tipo = partes[0]
        valores = list(map(float, partes[1:]))

        if tipo == "DENTRO":
            resposta = circulo_dentro(*valores)
        else:
            resposta = cabem_no_retangulo(*valores)

        print("S" if resposta else "N")


if __name__ == "__main__":
    main()
```

TÉCNICA

Contenção usa diferença dos raios; não sobreposição usa soma. Comparar quadrados evita sqrt.

ARMADILHA

Antes de elevar a diferença ao quadrado, teste qual raio é maior. Tangência usa <= ou >=.

Distância de ponto a segmento

Programa completo que projeta cada ponto no segmento e imprime a menor distância com duas casas.

Entrada: segmento e Q pontos. Saída: uma distância por ponto.

Tempo O(Q) · memória O(1)

EXEMPLO DE CÓDIGO

solucao.py

```python
import math


def distancia_ponto_segmento(p, a, b):
    px, py = p
    ax, ay = a
    bx, by = b
    vx = bx - ax
    vy = by - ay
    comprimento2 = vx * vx + vy * vy

    # Segmento degenerado: A e B são o mesmo ponto.
    if comprimento2 == 0.0:
        return math.hypot(px - ax, py - ay)

    # Projeção na reta, depois limitada ao segmento.
    t = ((px - ax) * vx + (py - ay) * vy) / comprimento2
    t = max(0.0, min(1.0, t))

    qx = ax + t * vx
    qy = ay + t * vy
    return math.hypot(px - qx, py - qy)


def main():
    ax, ay, bx, by = map(float, input().split())
    consultas = int(input())
    a = (ax, ay)
    b = (bx, by)

    for _ in range(consultas):
        ponto = tuple(map(float, input().split()))
        distancia = distancia_ponto_segmento(ponto, a, b)
        print(f"{distancia:.2f}")


if __name__ == "__main__":
    main()
```

TÉCNICA

t entre 0 e 1 usa a perpendicular; fora desse intervalo, a extremidade mais próxima vence.

ARMADILHA

Distância à reta infinita pode cair fora do segmento e produzir resposta impossível.

Circuncírculo e comparação com EPS

Programa completo que testa se todos os pontos pertencem ao círculo definido pelos três primeiros.

Entrada: N pontos. Saída: SIM, NAO ou COLINEARES.

Tempo O(N) · memória O(N)

EXEMPLO DE CÓDIGO

solucao.py

```python
import math


EPS = 1e-7


def circuncirculo(a, b, c):
    ax, ay = a
    bx, by = b
    cx, cy = c
    d = 2.0 * (
        ax * (by - cy)
        + bx * (cy - ay)
        + cx * (ay - by)
    )

    # Determinante zero: os três pontos são colineares.
    if math.isclose(d, 0.0, abs_tol=EPS):
        return None

    a2 = ax * ax + ay * ay
    b2 = bx * bx + by * by
    c2 = cx * cx + cy * cy
    ux = (a2 * (by - cy) + b2 * (cy - ay)
          + c2 * (ay - by)) / d
    uy = (a2 * (cx - bx) + b2 * (ax - cx)
          + c2 * (bx - ax)) / d
    raio2 = (ax - ux) ** 2 + (ay - uy) ** 2
    return ux, uy, raio2


def main():
    n = int(input())
    pontos = [tuple(map(float, input().split())) for _ in range(n)]
    circulo = circuncirculo(*pontos[:3])

    if circulo is None:
        print("COLINEARES")
        return

    ux, uy, raio2 = circulo
    todos = all(
        math.isclose(
            (x - ux) ** 2 + (y - uy) ** 2,
            raio2, rel_tol=EPS, abs_tol=EPS
        )
        for x, y in pontos
    )
    print("SIM" if todos else "NAO")


if __name__ == "__main__":
    main()
```

TÉCNICA

Três pontos não colineares determinam um círculo. O determinante detecta a degeneração.

ARMADILHA

Resultados de divisões não devem ser comparados com ==. Use tolerância relativa e absoluta.

Áreas, seno e conversão para radianos

Programa completo que atende consultas de áreas do quadrado com arcos e da razão pentágono-quadrado.

Entrada: T comandos AREA ou PENTAGONO. Saída: valores formatad...

Tempo O(T) · memória O(1)

EXEMPLO DE CÓDIGO

solucao.py

```python
import math


def areas(lado):
    quadrado = lado * lado
    complemento = quadrado - math.pi * quadrado / 4.0
    segmento = (
        quadrado * (4 * math.pi - 3 * math.sqrt(3)) / 24
    )
    a3 = 8 * segmento + 8 * complemento - 4 * quadrado
    a2 = 4 * complemento - 2 * a3
    a1 = quadrado - a2 - a3
    return a1, a2, a3


def lado_do_quadrado(lado_do_pentagono):
    # math.sin recebe radianos, não graus.
    return (
        lado_do_pentagono
        * math.sin(math.radians(108))
        / math.sin(math.radians(63))
    )


def main():
    t = int(input())

    for _ in range(t):
        tipo, texto = input().split()
        valor = float(texto)

        if tipo == "AREA":
            a1, a2, a3 = areas(valor)
            print(f"{a1:.3f} {a2:.3f} {a3:.3f}")
        else:
            print(f"{lado_do_quadrado(valor):.10f}")


if __name__ == "__main__":
    main()
```

TÉCNICA

Fórmulas constantes viram funções O(1). math.radians faz a conversão obrigatória antes de sin.

ARMADILHA

A ordem das áreas e a quantidade de casas pertencem ao enunciado. Não arredonde valores intermediários.

Backtracking com poda por limite

Programa completo que maximiza uma soma sem ultrapassar o limite, escolhendo, explorando e desfazendo.

Entrada: itens positivos e capacidade. Saída: maior soma possível.

Pior caso $O(2^n)$ · podas reduzem a busca

EXEMPLO DE CÓDIGO

solucao.py

```python
def melhor_soma(valores, limite):
    n = len(valores)
    sufixo = [0] * (n + 1)

    # Bound otimista: soma de tudo que ainda resta.
    for i in range(n - 1, -1, -1):
        sufixo[i] = sufixo[i + 1] + valores[i]

    melhor = 0

    def buscar(i, soma):
        nonlocal melhor

        if soma > limite:
            return  # ramo inviável

        melhor = max(melhor, soma)

        if i == n:
            return

        # Mesmo pegando tudo, este ramo não supera o incumbente.
        if soma + sufixo[i] <= melhor:
            return

        buscar(i + 1, soma + valores[i])  # escolher
        buscar(i + 1, soma)               # não escolher

    buscar(0, 0)
    return melhor


def main():
    n, limite = map(int, input().split())
    valores = list(map(int, input().split()))
    assert len(valores) == n
    print(melhor_soma(valores, limite))


if __name__ == "__main__":
    main()
```

TÉCNICA

Backtracking explora escolhas; branch and bound poda um ramo que não pode superar a melhor solução já encontrada.

ARMADILHA

O bound deve ser otimista e seguro. Um limite que subestima o potencial pode eliminar a resposta ótima.

Guloso com pilha monotônica

Programa completo do beecrowd 1084. Dígitos maiores removem dígitos menores anteriores.

Entrada: n, m e número até sentinela 0 0. Saída: maior número.

Tempo O(n) · memória O(n)

EXEMPLO DE CÓDIGO

solucao.py

```python
import sys

# buffer evita o custo de camadas extras de entrada em casos grandes.
entrada = sys.stdin.buffer

while True:
    n, m = map(int, entrada.readline().split())

    # O par 0 0 encerra a entrada.
    if n == 0 and m == 0:
        break

    numero = entrada.readline().strip().decode()

    # A pilha mantém o maior prefixo possível em ordem monotônica.
    pilha = []
    apagados = 0

    for digito in numero:
        # Um dígito maior substitui dígitos menores anteriores enquanto ainda
        # houver remoções disponíveis.
        while pilha and apagados < m and digito > pilha[-1]:
            pilha.pop()
            apagados += 1

        # A resposta precisa terminar com exatamente n - m algarismos.
        if len(pilha) < n - m:
            pilha.append(digito)

    print("".join(pilha))
```

TÉCNICA

A primeira posição diferente decide o maior número. Cada dígito entra e sai da pilha no máximo uma vez.

ARMADILHA

Uma estratégia gulosa precisa de justificativa. Escolha local sem prova pode falhar em outro problema.

DP de troco ilimitado

Programa completo do beecrowd 1034. mochila[j] guarda o menor número de blocos para formar j.

Entrada: tipos de bloco e comprimento alvo. Saída: mínimo de b...

Tempo O(N·M) · memória O(M)

EXEMPLO DE CÓDIGO

solucao.py

```python
N, M = 0, 0
mochila = []
blocos = []


def calculaMochila():
    # mochila[j] guarda o menor número de blocos que forma exatamente j.
    mochila = [-1 for _ in range(M + 1)]
    mochila[0] = 0

    for bloco in blocos:
        # O percurso crescente permite reutilizar o mesmo tamanho de bloco.
        for j in range(bloco, M + 1):
            if(mochila[j - bloco] != -1):
                if(mochila[j] == -1):
                    mochila[j] = mochila[j - bloco] + 1
                else:
                    mochila[j] = min(mochila[j], mochila[j - bloco] + 1)

    return mochila[M]


T = int(input())

for _ in range(T):
    N, M = map(int, input().strip().split(' '))
    # Os tamanhos disponíveis formam as transições da programação dinâmica.
    blocos = map(int, input().strip().split(' '))

    print(calculaMochila())
```

TÉCNICA

O laço crescente permite reutilizar o bloco atual. O estado j-bloco precisa ser alcançável.

ARMADILHA

Percorrer para trás resolveria a versão em que cada bloco só pode ser usado uma vez.

Mochila 0/1 com memória comprimida

Programa completo do beecrowd 1286. Cada pedido pode ser escolhido no máximo uma vez.

Entrada: pedidos com valor e peso. Saída: melhor valor na capa...

Tempo O(n·capacidade) · memória O(capacidade)

EXEMPLO DE CÓDIGO

solucao.py

```python
import sys

# O formato possui muitos inteiros; o buffer oferece leitura direta em bytes.
entrada = sys.stdin.buffer

while True:
    n = int(entrada.readline())

    if n == 0:
        break

    pizzas = int(entrada.readline())

    # dp[c] é o maior tempo economizado usando, no máximo, c pizzas.
    dp = [0] * (pizzas + 1)

    for _ in range(n):
        tempo, quantidade = map(int, entrada.readline().split())

        # Percorre de trás para frente porque cada pedido só pode ser escolhido
        # uma vez; avançar permitiria reutilizar o pedido na mesma rodada.
        for capacidade in range(pizzas, quantidade - 1, -1):
            dp[capacidade] = max(
                dp[capacidade],
                dp[capacidade - quantidade] + tempo
            )

    print(f"{dp[pizzas]} min.")
```

TÉCNICA

A capacidade é percorrida de trás para frente; assim o item atual não alimenta outro estado da mesma rodada.

ARMADILHA

A direção do laço define 0/1 ou ilimitada. A fórmula isolada não revela essa diferença.

Recursão e memoização de dois estados

Programa completo do beecrowd 1029. O cache guarda Fibonacci e a quantidade teórica de chamadas.

Entrada: consultas n. Saída: fib(n) e chamadas recursivas.

Tempo O(maior n) · memória O(maior n)

EXEMPLO DE CÓDIGO

solucao.py

```python
# F memoriza Fibonacci; CF memoriza o total de nós da árvore de chamadas.
F = [-1 for _ in range(40)]
CF = [-1 for _ in range(40)]

F[0] = 0
F[1] = 1

CF[0] = 1
CF[1] = 1

def calcula(n):
    # O estado só é expandido na primeira vez em que aparece.
    if(F[n] == -1):
        result1, num_calls1 = calcula(n - 1)
        result2, num_calls2 = calcula(n - 2)
        F[n] = result1 + result2
        # As duas subárvores mais 1 para a chamada calcula(n).
        CF[n] = num_calls1 + num_calls2 + 1
    return (F[n], CF[n])

# A memória permanece entre os casos e reaproveita resultados anteriores.
N = int(input())

for _ in range(N):
    X = int(input())
    result, num_calls = calcula(X)
    # O enunciado não conta a chamada inicial; por isso subtraímos 1.
    print(f'fib({X}) = {num_calls - 1} calls = {result}')
```

TÉCNICA

F[n] e CF[n] são estados distintos da mesma recorrência. Cada índice é calculado apenas uma vez.

ARMADILHA

Memoização evita recomputação, mas a versão top-down ainda usa pilha. A chave deve identificar todo o estado.