Relatório dissertativo da Atividade Avaliativa 1 (29/08/2025)

Objetivo do documento: comparar o desempenho dos códigos produzidos, a partir do exercício “Entrega 02”, do dia 29/08/2025, com as devidas especificações, sendo cobrados dois eixos de resolução: uma pela IA e outra, “caseira” (modo recursivo e iterativo). No caso, será discutido o desempenho final observado (notação “Big O”), além dos pontos nos quais foram contrastantes entre o desenvolvimento de cada caminho, para a resolução da situação problema em si.

Versão Iterativa)

Primeiramente, temos o código gerado pelo Sabiá-3.1, da Maritaca AI:

```c
#include <stdio.h>

int main() {
    int n, k, resultado = 1;

    scanf("%d", &n);
    scanf("%d", &k);

    for (int i = 0; i < k; i++) {
        resultado = resultado*n%1000;
    }

    // Imprime o resultado com três dígitos, adicionando zeros à esquerda se necessário
    printf("%d\n", resultado);

    return 0;
}
```

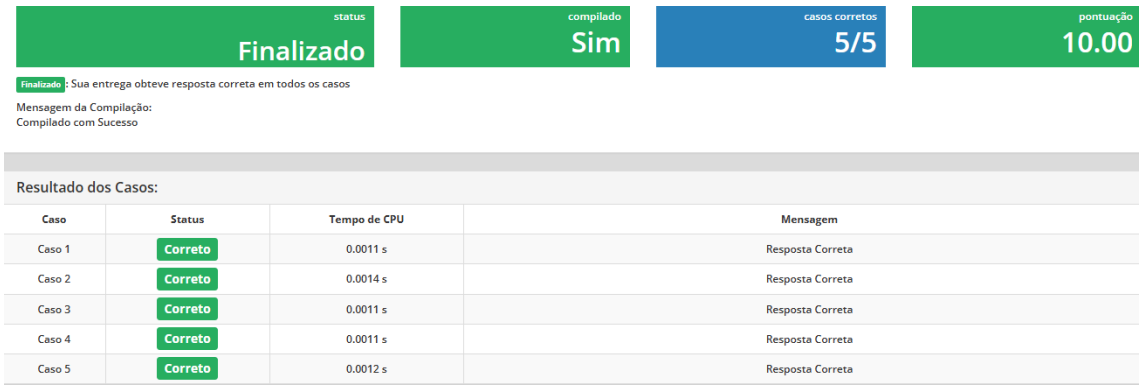
Em seguida, temos a versão feita pelo programador:

```c
#include <stdio.h>

int main() { // O maior valor aramazenável pelo tipo int vale 2³¹ - 1.
    int n, k, resultadoFinal = 1;

    scanf("%d %d", &n, &k);

    while (k > 0) { // Loop para "reaproveitar" os 3 últimos algarismos de cada número, otimizando a mudança de expoente.
        if (k%2 != 0) // Verifica se o expoente é par ou ímpar.
            resultadoFinal = (resultadoFinal*n)%1000; /* Garante que será o próprio número de uso de cada iteração,
            com resultadoFinal = 1 (garante a exponenciação ímpar). */

        n = (n*n)%1000; // A base é elevada ao quadrado e os 3 últimos dígitos são usados.
        k /= 2; // Expoente dividido pela metade.
    }

    printf("%d", resultadoFinal);

    return 0;
}
```

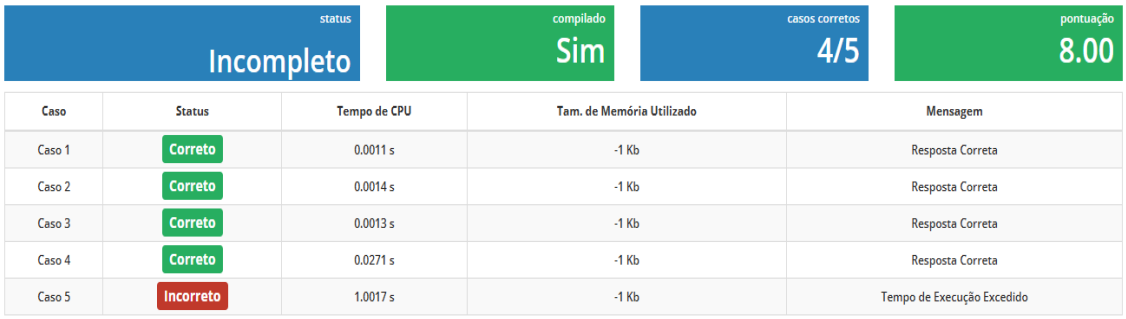
Discussão técnica: o modelo desenvolvido pela IA apresentou uma alternativa a qual envolveu a visão “padrão” para se resolver o problema apresentado: ilustrar os 3 últimos dígitos após k potenciações. Houve um foco no cálculo no formato mod(1000)” do número em exponenciação para cada iteração do bloco “for” gerando, assim, uma quantidade de comparações notavelmente maior no total,

sob uma perspectiva exata do total de operações que ocorreriam durante a execução do programa. Nota-se, portanto, uma natureza de problema escalável pela perspectiva de execução de tempo O(k), uma vez que os cálculos não são retrocedidos e muito menos reaproveitados ao longo da cadeia de cálculos sequenciais.

Já o código feito pelo programador, deu ênfase em um viés de otimização da operação de exponenciação, dado que envolveu o raciocínio de elevar ao quadrado a base do número a ser calculado o "mod(1000)", envolvendo um simples armazenamento final do resultado diretamente no "if", moldado especificamente para expoentes ímpares. Ou seja, mesmo que o expoente seja ímpar, logo de início, a lógica da exponenciação quadrática é preservada, assim como a sua eficiência prática. Percebe-se uma guinada notável para as propriedades matemáticas da exponenciação, a fim de simplificar o total de cálculos necessários para a determinação do resultado. Nesse caso, percebe-se que a natureza do problema é O(log[k]).



Tempo de execução do método iterativo (versão "caseira")



Tempo de execução do método iterativo (versão da IA)

Versão Recursiva)

A versão desenvolvida, em questão, é esta:

```c
#include <stdio.h>

int recursaoDividirParaConquistar(int base, int expoente) {
    if (expoente == 0)
        return 1;

    else if (expoente == 1)
        return base%1000;

    else {
        if (expoente%2 == 0){
            int medio = recursaoDividirParaConquistar(base, expoente/2);
            return (medio*medio)%1000;
        }

        else {
            int medio = recursaoDividirParaConquistar(base, expoente/2);
            return (base*(medio*medio%1000))%1000; // Colocando a repetição faltante da base
        }
    }
}

int main() { // O maior valor aramazenável pelo tipo int vale 2³¹ - 1.
    int n, k;

    scanf("%d %d", &n, &k);

    if (n > 99 || n < 0 || k < 0 || k > 1000000000)
        return 0;

    if (k == 1)
        printf("%d", n);

    else
        printf("%d", recursaoDividirParaConquistar(n, k));

    return 0;
}
```

Discussão técnica: é visível o gasto de memória o qual o método recursivo proporciona, já que depende de chamadas em cadeia de uma função qualquer sobre si mesma, sendo de natureza O(k) para o problema apresentado, já que dependeria exatamente do valor de k, o qual é o expoente do número a ser calculado, sendo "n" a base. Para o contexto do problema, se percebe, certa diferença se comparado com o método iterativo no tempo de execução, apesar de ambos, na notação "Big O", serem algoritmos de desempenho O(k).

status	compilado	casos corretos	pontuação
Finalizado	Sim	5/5	10.00

Caso	Status	Tempo de CPU	Tam. de Memória Utilizado	Mensagem
Caso 1	Correto	0.0019 s	-1 Kb	Resposta Correta
Caso 2	Correto	0.0011 s	-1 Kb	Resposta Correta
Caso 3	Correto	0.0011 s	-1 Kb	Resposta Correta
Caso 4	Correto	0.0012 s	-1 Kb	Resposta Correta
Caso 5	Correto	0.0011 s	-1 Kb	Resposta Correta

Tempo de execução do método recursivo

Discussões Comparativas)

Observou-se que a IA entregou uma resolução direta para o problema, de desempenho O(k), não considerando, necessariamente, a resolução mais otimizada possível para o contexto solicitado em “Entrega 02”, sendo isso visível no cálculo adotado para o ciclo de iteração propriamente adotado pelo algoritmo em si. O chat imitou, realmente, o pensamento mais comum para solucionar, supostamente, o problema do overflow de memória para a representação de variáveis inteiras do C (o maior valor armazenável é igual a 2^{31} - 1), porém o raciocínio evidentemente falha para expoentes consideravelmente grandes, como os valores perto do limite de 10^9. Isso se deve ao fato de o cálculo ser feito de forma sequencialmente multiplicativa, “percorrendo” o número n^k, k vezes, com gradual ocupação de uma mesma memória, acarretando, eventualmente, no estouro mencionado.

Já a versão “caseira”, ela mostrou uma alternativa mais viável, pois detém um desempenho notório (O(log[k])) ao aplicar o raciocínio da exponenciação com “saltos” de valores (exponenciação quadrática), resumindo cálculos que, para valores de k grandes, são mais onerosos, além de não sobrecarregar o swapping da heap, durante a execução do código. Dessa forma, tem-se um processo exponencial sendo efetivado com um dinamismo logarítmico, garantindo que não haverá o problema do overflow, resolvendo, efetivamente, o problema requisitado.

A versão recursiva, para esse problema, apresentou um desempenho notoriamente melhor do que a iterativa, mesmo gastando, visivelmente, mais memória, sendo esse justamente o fator para o ganho. Como aquela versão utiliza mais área de memória, o swapping terá mais endereços para atuar, garantindo que, independentemente do fato do código, ao final, ser maior do que a versão iterativa intuitiva, o código possa ser executado sem o problema do “runtime error”. Pelo fato de estarmos em um ambiente de resolução controlado, ou seja, os valores da base e do expoente máximos são limitados, a memória não será um problema a ser relevado, entretanto, para aplicações mais genéricas, a versão recursiva também trará problemas, evidenciando que a versão logarítmica (“caseira”) é a ideal como resposta.