Universidade Federal do Pará
Instituto de Ciências Exatas e Naturais
Faculdade de Computação
Análise de Algoritmos

Lista de Exercícios
Fundamentos

1. Sejam f, g e h funções reais positivas da variável inteira n. Com respeito às notações assintóticas, avalie as afirmativas abaixo.

I. $f = O(g)$ se e somente se $g = \cancel{\omega}(f)$. Ω

II. $f = \Theta(g)$ se e somente se $f = O(g)$ e $f = \Omega(g)$. → ou $g = O(f)$

III. $f = o(g)$ e $g = o(h)$ implicam $f = o(h)$.

IV. $\cancel{n}log(n) + k = O(n)$, dado que k é uma constante positiva.

A análise permite concluir que somente

(A) a afirmativa III é verdadeira.
(B) as afirmativas I e IV são verdadeiras.
(C) as afirmativas II e III são verdadeiras.
(D) as afirmativas I, II e III são verdadeiras.
(E) as afirmativas II, III e IV são verdadeiras.

lim. inf. [Ω] p/ o pior caso
↓
o melhor do pior caso

2. Um limite inferior para um problema P é uma função f, tal que a complexidade no tempo de pior caso de qualquer algoritmo que resolva P é $\Omega(f)$. Isto quer dizer que

(A) todo algoritmo que resolve P efetua $\Theta(f)$ passos.
(B) se existir um algoritmo A que resolve P com complexidade $O(f)$, então A é denominado algoritmo ótimo para P.
(C) todo algoritmo que resolve P é recursivo.
(D) todo algoritmo que resolve P é interativo.
(E) todo algoritmo que resolve P tem complexidade linear no mínimo.

3. O algoritmo para calcular um polinômio $a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x + a_0$, quando $x = c$, pode ser expresso em pseudocódigo por:

```
 Polinomio(A, n, c)
 power = 1                        1
 y = A[0]                         1
 para i = 1 até n faça            n+1
     power = power * c            n
     y = y + A[i] * power         n
 retornar (y)                     1
```

$T(n) = 1 + 1 + n + 1 + n + n + 1$

$= \cancel{4} + \cancel{3}n$

$T(n) = O(n) \rightarrow$ mas ele quer em Little-o

Seja $T(n)$ o tempo de execução do algoritmo acima descrito para as entradas $A[a_0...a_n]$, n e c. A ordem de $T(n)$ usando a notação Little-o é

Logo,

$T(n) = o(n^2)$

(A) $T(n) = o(c)$.
(B) $T(n) = o(log(n))$.
(C) $T(n) = o(\sqrt{n})$.
(D) $T(n) = o(n)$.
(E) $T(n) = o(n^2)$.

4. Considere o algoritmo abaixo.

```
 PROC(n)
 se n <= 1 então                        } Passo base
     retornar (2)                       }
 senão                                  { Passo reconrente
     retornar (PROC(n/2) + PROC(n/2))   {
 fim se                                 {
```

$$\begin{cases} T(1) = 2 \\ T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2}\right) \end{cases} \quad K = 1$$

$T(n) = 2\left[2T\left(\frac{\frac{n}{2}}{2}\right)\right] = 4T\left(\frac{n}{4}\right) \quad K = 2$

$T(n) = 2\left[4T\left(\frac{\frac{n}{4}}{2}\right)\right] = 8T\left(\frac{n}{8}\right) \quad K = 3$

Assinale a alternativa que indica o valor retornado pelo algoritmo considerando a entrada $n = 64$.

(A) 128.
(B) 130.
(C) 1.024.
(D) 4.096.
(E) 4.160.

Conjecturando...

$T(n) = 2^K T\left(\frac{n}{2^K}\right)$

Passo base...

$\frac{n}{2^K} = 1$

$n = 2^K$

$\log n = \log 2^K$

$\log n = K$

$T(n) = 2^{\log n} T\left(\frac{n}{2^{\log n}}\right)$

$T(n) = n \cdot T\left(\frac{n}{n}\right)$

$T(n) = n \cdot T(1)$

$T(n) = n \cdot 2$

Logo,

$T(64) = 64 \cdot 2 = 128$

5. O tempo de execução $T(n)$ de um algoritmo, em que n é o tamanho da entrada, é dado pela equação de recorrência $T(n) = 8T(n/2) + qn$ para todo $n > 1$. Dado que $T(1) = p$, e que p e q são constantes arbitrárias, a complexidade do algoritmo é

(A) $\Theta(log(n))$.
(B) $\Theta(n)$.
(C) $\Theta(nlog(n))$.
(D) $\Theta(n^2)$.
(~~E~~) $\Theta(n^3)$.

Teorema Mestre
$a = 8$
$b = 2$
$f(n) = qn$
$\log_b a = \log_2 8 = 3$

$C_1: qn = O(n^{3-\varepsilon})$ ✓
↓ Logo,
$T(n) = \Theta(n^3)$

6. Resolva as relações de recorrência abaixo pelo Teorema Mestre.

(a) $T(1) = 1$.
$T(n) = 4T(n/4) + n$ para $n > 1$.

$a = 4$
$b = 4$
$f(n) = n$
$\log_4 4 = 1$
$C_2: n = \Theta(n^1)$ ✓
↓ Logo,
$T(n) = \Theta(n \log n)$

(b) $T(1) = 1$.
$T(n) = 7T(n/2) + n^2$ para $n > 1$.

$a = 7$
$b = 2$
$f(n) = n^2$
$\log_b a = \log_2 7 \rightarrow 2 < \log_2 7 < 3$
$C_1: n^2 = O(n^{\log_2 7 - \varepsilon})$
↓ Logo,
$T(n) = \Theta(n^{\log_2 7})$

(c) $T(1) = 1$.
$T(n) = 3T(n/9) + 2n$ para $n > 1$.

$a = 3$
$b = 9$
$f(n) = 2n$
$\log_b a = \log_9 3 = 0.5$
$\log_9 3 = x \rightarrow 9^x = 3 \rightarrow 3^{2x} = 3$, $2x = 1$, $x = \frac{1}{2}$
$C_3: 2n = \Omega(n^{0.5+\varepsilon})$
↓ Logo,
$T(n) = \Theta(n)$

7. As definições recursivas apresentadas abaixo descrevem o tempo de execução de dois algoritmos recursivos: A e B:

$T_A(1) = 1$ e $T_A(n) = 2T_A(n-1) + 2$ para $n \geq 2$.

$T_B(1) = 1$ e $T_B(n) = T_B(n/2) + n$ para $n \geq 2$.

Assinale a alternativa correta.

(A) Os algoritmos não são eficientes. ✗
(B) Os algoritmos são assintoticamente equivalentes. ✗
(C) O algoritmo B tem complexidade exponencial no tempo. ✗
(D) O algoritmo A é mais eficiente assintoticamente que o algoritmo B. ✗
(~~E~~) O algoritmo B é mais eficiente assintoticamente que o algoritmo A. ✓

T. Mestre
$a = 1$
$b = 2$
$f(n) = n$
$\log_2 1 = 0$
$C_3:$
$n = \Omega(n^{0+\varepsilon})$
↓ Logo,
$T_B(n) = \Theta(n)$

$T_A(n) = 2T_A(n-1) + 2$ $K = 1$
$T_A(n) = 2[2T_A(n-1-1) + 2] + 2$ $K = 2$
$= 4T_A(n-2) + 6$
$T_A(n) = 2[4T_A(n-2-1) + 6] + 2$
$= 8T_A(n-3) + 14$
Conjecturando...
$T_A(n) = 2^K T_A(n-K) + 2^K - 2$

Passo base...
$n - K = 1$
$K = n - 1$
$T_A(n) = 2^{n-1} + 2^{n-1} - 2$
$T_A(n) = 2 \cdot 2^{n-1} - 2$
$= 2^n - 2$
$= \Theta(2^n)$

8. Considere o algoritmo A abaixo.

```
Algoritmo A(n)
Entrada: n, inteiro, n > 0.
{
   se n = 1
        retornar (1);
   senão
        retornar (2 * A(n /2) + 1);
}
```

P.B. (se n = 1 / retornar (1);)

P.R. (senão / retornar (2 * A(n /2) + 1);)

$T(1) = 1$

$T(n) = \Theta(1).T\left(\frac{n}{2}\right) + \Theta(1) = T\left(\frac{n}{2}\right) + 1$

T. Mestre

$a = 1$

$b = 2$

$f(n) = 1$

$\log_b^a = \log_2^1 = 0$

C_2:

$1 = \Theta(n^0)$

$1 = \Theta(1)$

$\downarrow$ Logo,

$T(n) = \Theta(n^0 . \log n) = \Theta(\log n)$

A complexidade no tempo de pior caso do algoritmo A é

(A) linear.
(B) logarítmica.
(C) quadrática.
(D) exponencial.
(E) $nlog(n)$.

9. O algoritmo recursivo abaixo soma os n primeiros números naturais.

```
Algoritmo Soma(n)
Entrada: n, inteiro, n > 0.
{
   se n = 1
        retornar (1);
   senão
        retornar (Soma(n - 1) + n);
}
```

P.B. (se n = 1 / retornar (1);)

P.R. (senão / retornar (Soma(n - 1) + n);)

$T(1) = 1$

$T(n) = T(n-1) + \Theta(1) = T(n-1) + 1 \quad K = 1$

$T(n) = [T(n-1-1)+1] + 1 = T(n-2) + 2 \quad K = 2$

$T(n) = [T(n-2-1)+2] + 1 = T(n-3) + 3 \quad K = 3$

Conjecturando...

$T(n) = T(n-K) + K$

Passo Base...

$n - K = 1$

$K = n - 1$

$T(n) = 1 + n - 1 = n \rightarrow \Theta(n)$

A complexidade no tempo do algoritmo é

(A) linear.
(B) logarítmica.
(C) quadrática.
(D) exponencial.
(E) $nlog(n)$.

10. Suponha que f e g são funções reais positivas da variável inteira n. Verifique se as seguintes afirmações são verdadeiras ou falsas. Justifique sua resposta caso a afirmativa seja falsa.

(a) Se $f(n) = O(g(n))$, então $2^{f(n)} = O(2^{g(n)})$. Falso, ex.: $f(n) = 2n$ e $g(n) = n$

(b) $f(n) = O((f(n))^2)$. Falso, ex.: $f(n) = \frac{1}{n}$

(c) Se $f(n) = O(g(n))$, então $g(n) = \Omega(f(n))$. Verdadeira pior caso

(d) $f(n) + O(f(n)) = \Theta(f(n))$. Verdadeiro, pois $f(n) + O(f(n)) = 2f(n) = \Theta(f(n))$

11. Escreva um algoritmo (em pseudocódigo) que realize busca binária de forma iterativa e o implemente numa linguagem de programação a sua escolha. Construa um gráfico mostrando a relação valor de entrada x tempo de execução do algoritmo implementado. Considerando uma análise assintótica em pior caso, explique se o desempenho do algoritmo implementado é superior, inferior ou igual ao do algoritmo que implementa busca binária de forma recursiva.

Para as questões **12** e **13**, entregue os seguintes itens considerando o algoritmo implementado para resolver o problema computacional:

- Uma captura de tela que mostre a compilação correta na plataforma de teste;
- O cálculo da complexidade no tempo usando notação assintótica; e
- Um gráfico ilustrando a análise empírica, ou seja, a relação valor de entrada x tempo de execução.

12. Resolva o seguinte problema computacional:

Problema: Ajude a Federação (#1588)
https://www.beecrowd.com.br/judge/pt/problems/view/1588

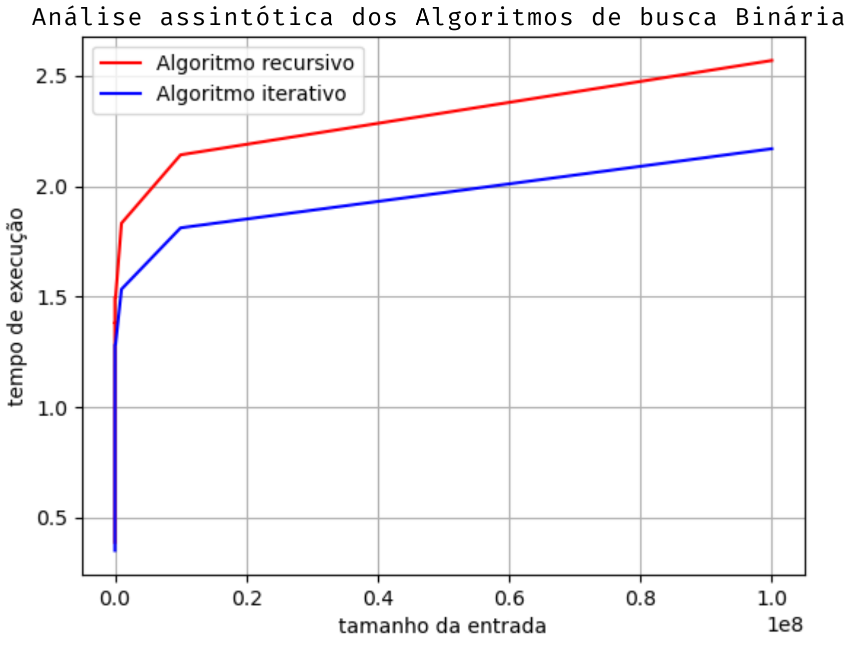
13. Resolva o seguinte problema computacional de forma **recursiva**:

Problema: A Lenda de Flavious Josephus (#1030)
https://www.beecrowd.com.br/judge/pt/problems/view/1030

11. Escreva um algoritmo (em pseudocódigo) que realize busca binária de forma iterativa e o implemente numa linguagem de programação a sua escolha. Construa um gráfico mostrando a relação valor de entrada x tempo de execução do algoritmo implementado. Considerando uma análise assintótica em pior caso, explique se o desempenho do algoritmo implementado é superior, inferior ou igual ao do algoritmo que implementa busca binária de forma recursiva.

```
BINARY-SEARCH(listaOrdenada, valorProcurado, menor, maior):
    indiceDoElementoDoMeio = (menor + maior) // 2
    elementoDoMeio = listaOrdenada[indiceDoElementoDoMeio]

    encontrado = False
    Enquanto não encontrado faça:
        Se (menor = maior) então:
            Se (elementoDoMeio ≠ valorProcurado) então:
                Retorne Nulo
            Senão
                encontrado = True
        Senão Se (elementoDoMeio = valorProcurado) então:
            encontrado = True
        Senão Se (elementoDoMeio > valorProcurado) então:
            novaPosicao = indiceDoElementoDoMeio - 1
            maior = novaPosicao
            indiceDoElementoDoMeio = (menor + maior) // 2
            elementoDoMeio = listaOrdenada[indiceDoElementoDoMeio]
            Se (elementoDoMeio = valorProcurado) então:
                encontrado = True
        Senão Se (elementoDoMeio < valorProcurado) então:
            novaPosicao = indiceDoElementoDoMeio + 1
            menor = novaPosicao
            indiceDoElementoDoMeio = (menor + maior) // 2
            elementoDoMeio = listaOrdenada[indiceDoElementoDoMeio]
            Se (elementoDoMeio = valorProcurado) então:
                encontrado = True
    Retorne indiceDoElementoDoMeio
```



Conforme a análise empírica dos dois algoritmos, percebe-se que ambos apresentam complexidade logarítmica, ou seja, assintoticamente eles são praticamente iguais. No entanto, aparentemente, o algoritmo iterativo apresenta um desempenho um pouco melhor por utilizar loops simples, já o algoritmo recursivo, por fazer múltiplas chamadas de função, ele precisa alocar espaço para as variáveis locais e empilhar argumentos, causando um pequeno overhead comparado à execução de um loop simples.

Para as questões **12** e **13**, entregue os seguintes itens considerando o algoritmo implementado para resolver o problema computacional:

- Uma captura de tela que mostre a compilação correta na plataforma de teste;
- O cálculo da complexidade no tempo usando notação assintótica; e
- Um gráfico ilustrando a análise empírica, ou seja, a relação valor de entrada x tempo de execução.

12. Resolva o seguinte problema computacional:

Problema: Ajude a Federação (#1588)
https://www.beecrowd.com.br/judge/pt/problems/view/1588

SUBMISSÃO # 35749672

PROBLEMA:	1588 - Ajude a Federação
RESPOSTA:	Accepted
LINGUAGEM:	C++17 (g++ 7.3.0, -std=c++17 -O2 -lm) [+0s]
TEMPO:	0.029s
TAMANHO:	2,79 KB
MEMÓRIA:	-
SUBMISSÃO:	24/09/2023 18:48:48

CÓDIGO FONTE

```
#include <iostream>
#include <string>
#include <map>
#include <cmath>
#include <vector>

using namespace std;

typedef struct Time{
    string nomeDoTime;
    int pontos;
    int vitorias;
    int gols;
    int ordem;
}
Time;

bool comparaTimes(const Time& time1, const Time& time2){
    if (time1.pontos != time2.pontos) return time1.pontos > time2.pontos;
    if (time1.vitorias != time2.vitorias) return time1.vitorias > time2.vitorias;
    if (time1.gols != time2.gols) return time1.gols > time2.gols;
    return time1.ordem < time2.ordem;
}

int main() {

    int T, N, M, gols1, gols2;
    string time1, time2, nomeDoTime;


    cin >> T;
    for (int i = 0; i < T; i++){
        map<string, Time> listaDeClassificacao;
        cin >> N >> M;

```

$O(T)$

```cpp
        for (int i = 0; i < N; i++){
            // cout << "Nome do time: \n";
            cin >> nomeDoTime;
            Time time;
            time.nomeDoTime = nomeDoTime;
            time.gols = 0;
            time.pontos = 0;
            time.vitorias = 0;
            time.ordem = i;
            listaDeClassificacao[nomeDoTime] = time;
        }
```

→ $O(N)$

```cpp
        for (int i = 0; i < M; i++){
            cin >> gols1 >> time1 >> gols2 >> time2;
            if (gols1 - gols2 > 0){
                listaDeClassificacao[time1].gols += gols1;
                listaDeClassificacao[time1].pontos += 3;
                listaDeClassificacao[time1].vitorias += 1;
                listaDeClassificacao[time2].gols += gols2;
            }else if (gols1 - gols2 < 0){
                listaDeClassificacao[time2].gols += gols2;
                listaDeClassificacao[time2].pontos += 3;
                listaDeClassificacao[time2].vitorias += 1;
                listaDeClassificacao[time1].gols += gols1;
            }else{
                listaDeClassificacao[time2].gols += gols2;
                listaDeClassificacao[time2].pontos += 1;
                listaDeClassificacao[time1].gols += gols1;
                listaDeClassificacao[time1].pontos += 1;
            }
        }
```

→ $O(M)$

```cpp
        vector<Time> timesOrdenados;
        for (auto& par : listaDeClassificacao) {
            timesOrdenados.push_back(par.second);
        }
```

→ $O(N)$

↳ listaDeClassificacao tem o tamanho da quantidade de times N

```cpp
        int tamanho = timesOrdenados.size();
```

→ também tem tamanho N

```cpp
        for (int i = 0; i < tamanho-1; i++) {
            for (int j = 0; j < tamanho-i-1; j++) {
                if (!(comparaTimes(timesOrdenados[j], timesOrdenados[j+1]))) {
                    Time aux = timesOrdenados[j];
                    timesOrdenados[j] = timesOrdenados[j+1];
                    timesOrdenados[j+1] = aux;
                    // swap(arr[j], arr[j+1]);
                }
            }
        }
```

Bubble Sort

$O(N^2)$

```cpp
        for (int i = 0; i < tamanho; i++){
            cout << timesOrdenados[i].nomeDoTime << "\n";
        }
    }

    return 0;
}
```

$O(N)$

Como o código está todo dentro do Loop for que itera até T, então a complexidade vai ser dada por:

$$O(T) \cdot O(N + M + N + N^2 + N)$$

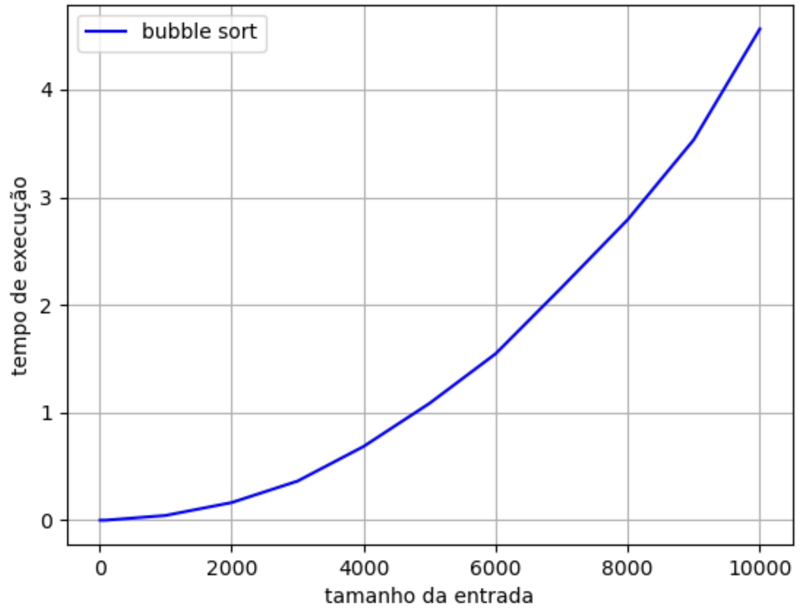
$$O(T) \cdot O(N^2 + \cancel{3}N + M)$$

$$O(T) \cdot O(N^2 + N + M)$$

$$O(T \cdot (N^2 + N + M))$$

Considerando que a parte dominante do algoritmo é a ordenação dos times, podemos simplificar a complexidade para: $O(N^2)$

Análise assintótica da ordenação dos times usando bubble sort



13. Resolva o seguinte problema computacional de forma **recursiva**:

Problema: A Lenda de Flavious Josephus (#1030)
https://www.beecrowd.com.br/judge/pt/problems/view/1030

SUBMISSÃO # 35757891

PROBLEMA:	1030 - A Lenda de Flavious Josephus
RESPOSTA:	Accepted
LINGUAGEM:	C++17 (g++ 7.3.0, -std=c++17 -O2 -lm) [+0s]
TEMPO:	0.221s
TAMANHO:	1,63 KB
MEMÓRIA:	-
SUBMISSÃO:	25/09/2023 09:03:21

CÓDIGO FONTE

```cpp
#include <iostream>
#include <cstdlib>

using namespace std;

typedef struct pessoa *Circulo;

struct pessoa{
    struct pessoa *prox;
    int numeracao;
    int salto;
};

Circulo cria_pessoa(int numeracao, int salto){
    Circulo novaPessoa;
    novaPessoa = (struct pessoa *)malloc(sizeof(struct pessoa));
    novaPessoa->prox = novaPessoa; // Inicializa o próximo como ele mesmo
    novaPessoa->numeracao = numeracao;
    novaPessoa->salto = salto;

    return novaPessoa;
}
```

O(1)

```cpp
void add_pessoa(Circulo ultimo, int numeracao, int salto){
    Circulo novaPessoa = cria_pessoa(numeracao, salto);
    Circulo proxUltimo = ultimo->prox;

    novaPessoa->prox = proxUltimo;
    ultimo->prox = novaPessoa;
}
```

O(1)

```cpp
void remover_pessoa(Circulo circ){
    Circulo pessoaRemovida = circ->prox;
    circ->prox = pessoaRemovida->prox;
    free(pessoaRemovida);
}
```

O(1)

```cpp
int main (){
```

```
    int NC;
    int n;
    int k;
    int numeracaoFinal;
    cin >> NC;

    for (int i = 0; i < NC; i++){
        cin >> n >> k;

        Circulo circulo = cria_pessoa(1, k); // Começando a numeracao de 1

        for (int i = 2; i <= n; i++){
            add_pessoa(circulo, i, k);
            circulo = circulo->prox;
        }

        while (circulo->prox != circulo) {
            for (int i = 1; i < k; i++) {
                circulo = circulo->prox;
            }
            remover_pessoa(circulo);
        }

        cout << "Case " << i+1 << ": " << circulo->numeracao << "\n";
        free(circulo); // Liberando a última pessoa

    }

    return 0;
}
```

$\rightarrow O(NC + 1) \rightarrow O(NC)$

$\rightarrow O(1)$

$\rightarrow O(n)$

leva n iterações para eliminar todos do círculo, até restar somente 1

$O(n.K)$

leva k iterações para percorrer o círculo de k em k

$\rightarrow O(1)$

$\rightarrow O(1)$

Como o código está dentro do loop for que itera até NC, a complexidade do algoritmo vai ser dada por:

$$O(NC) . O(\cancel{1} + \cancel{n} + n.K + \cancel{1} + \cancel{1})$$

$$O(NC) . O(n.K)$$

$$O(NC . n.K)$$

Como a parte dominante do código é a adição e remoção de pessoas no círculo e o loop for que itera até NC é apenas para aumentar os casos de teste por execução do código, podemos simplificar a complexidade para:

$$O(n + n.K)$$

$$O(nK)$$

Análise assinótica da Lenda de Flavious Josephus