Algoritmos e Complexidade
Ficha 2: Resolução

Eduardo Freitas Fernandes

2025

1 Contagem

__**Exercício 1**__

Função `bubbleSort()`:

Comparações entre elementos do array: O número de comparações entre elementos do array é igual em todos os casos, isto é, o conteúdo do array não altera a execução da função, logo o melhor e pior caso terão o mesmo custo.

$$T_{bubbleSort}(N) = \sum_{i=1}^{N-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{N-1} i = \frac{(N-1) \times N}{2} = \Theta(N^2)$$

Trocas Efetuadas:

- **Melhor caso**: array ordenado por ordem crescente (condição do `if statement` é sempre falsa)
- **Pior caso**: array ordenado por ordem decrescente (condição do `if statement` é sempre verdadeira)

Função `iSort()`:

Comparações entre elementos do array:

- **Melhor caso**: array ordenado por ordem crescente (apenas uma comparação no loop interno, de cada iteração do loop externo)

$$T_{iSort}(N) = \sum_{i=1}^{N-1} 1 = N - 1 = \Omega(N)$$

- **Pior caso**: array ordenado por ordem decrescente

$$T_{iSort}(N) = \sum_{i=1}^{N-1} \sum_{j=1}^{i} 1 = \sum_{i=1}^{N-1} i = \frac{N^2 - N}{2} = O(N^2)$$

Para as trocas de elementos, o melhor e pior caso são os mesmos, dado que o `swap` depende da condição de paragem do `for loop`.

Exercício 2

Seja N o número de bits necessários para representar `x`, teremos a seguinte gama de valores:

- 2^{N-1}, que corresponde a todos os bits excepto o mais significativo serem 0 (melhor caso)
- $\sum_{k=0}^{N-1} 2^k = 2^N - 1$, no caso dos bits serem todos 1 (pior caso)

Função `mult1()`:

O número de somas e subtrações é igual, logo não é necessário especificar cada um. O número de operações primitivas (`+ -`) no **pior caso** será:

$$T_{mult1}(N) = 2^N - 1 = \Theta(2^N)$$

Função `mult2()`:

No pior caso, todos os bits estão a 1, logo a condição avaliada no `if statement` será sempre verdadeira, então a operação primitiva `+` será executada tantas vezes quanto as operações `% / *`. Podemos concluir que a complexidade é **linear**, em função do número de bits, pois a cada iteração um dos bits de `x` passa a 0 e é efetuada uma adição.

Exercício 3

$$T(N) = \sum_{i=0}^{N-1} \sum_{j=i}^{N-1} \sum_{k=i}^{j} 1 = \sum_{i=0}^{N-1} \sum_{j=i}^{N-1} j - i + 1 = \sum_{i=0}^{N-1} \sum_{j=i}^{N-1} j$$

$$= \sum_{i=0}^{N-1} \frac{(N-i) \times (N-1+i)}{2} = \sum_{i=0}^{N-1} \frac{N^2 + ...}{2} = \frac{N^3}{3} + ... = \Theta(N^3)$$

Nota: na resolução deste somatório decidi remover a expressão $-i+1$ devido ao facto de ser uma constante no segundo somatório, então teria pouco impacto no resultado final da complexidade e iria dificultar os cálculos.

```
int maxSoma(int v[], int N) {
    int max, i, t;
    int c[N];
    c[0] = v[0];
    max = c[0];
    for (i = 1; i < N; i++) {
        t = c[i - 1] + v[i];

        if (t > c[i - 1]) c[i] = t;
        else c[i] = v[i];

        if (c[i] > max) max = c[i];
    }
    return max;
}
```

$$T_{maxSoma}(N) = 1 + \sum_{i=1}^{N-1} 2 = 1 + 2 \times (N-1) = \Theta(N)$$

Exercício 4

Comparações entre elementos do array:

- **Melhor Caso**: array estritamente decrescente
- **Pior Caso**: array estritamente crescente

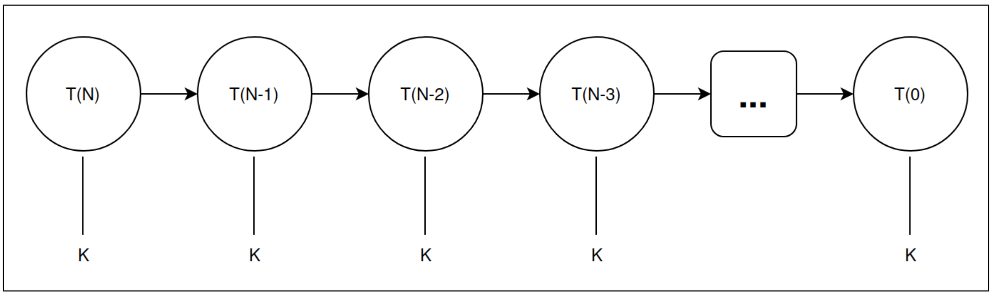
$$T_{maxcresc}(N) = \sum_{i=0}^{N-1} \sum_{j=1}^{N-1} 1 = \sum_{i=0}^{N-1} N - 1 = N \times (N-1) = O(N^2)$$

Ao realizar a optimização `i += m`, serão efetuadas apenas `N` comparações, pois `m` terá o valor `N`, pondo fim ao ciclo. Isto acontece porque o array está ordenado por ordem crescente, então `m` será o comprimento do maior segmento crescente, que corresponde a `N`.

2 Definições Recursivas

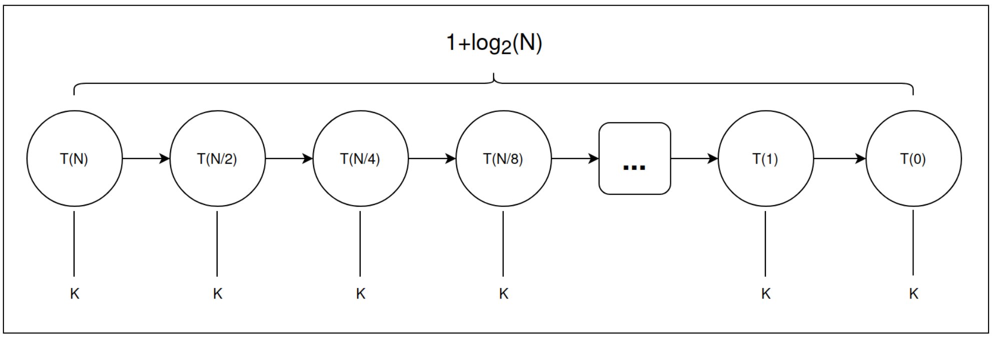
Exercício 1

a)



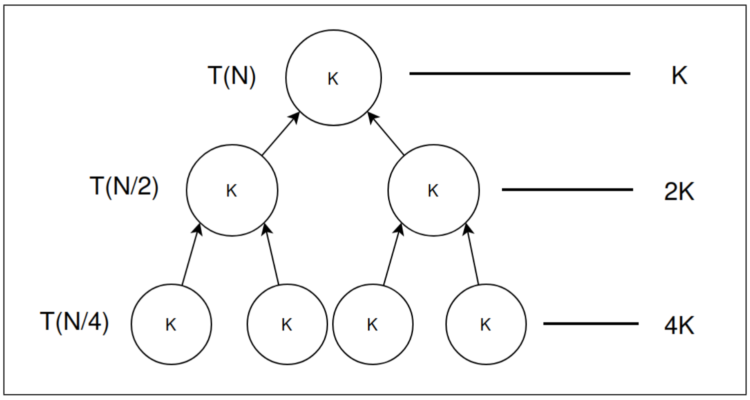
$$T(N) = \sum_{i=0}^{N} K = (N+1) \times K = \Theta(N)$$

b)



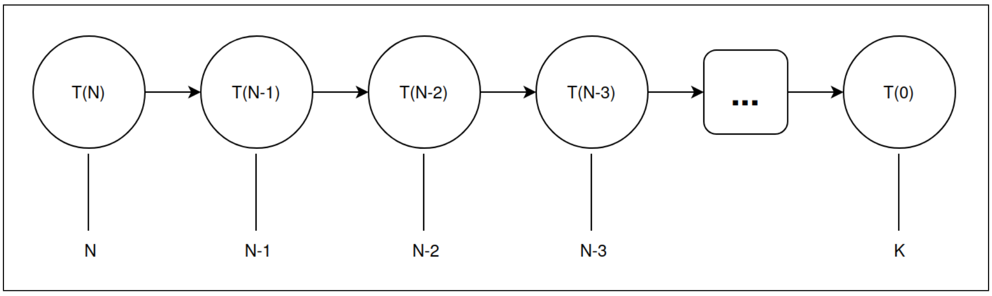
$$T(N) = \sum_{i=0}^{1+\log_2(N)} K = (2 + \log_2(N)) \times K = \Theta(\log_2(N))$$

c)



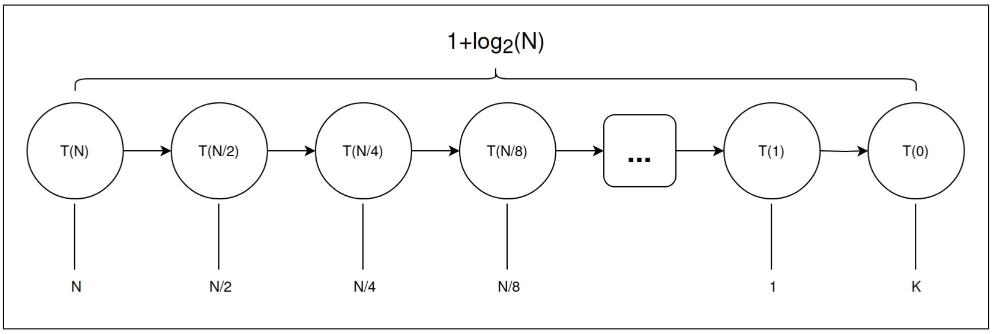
$$T(N) = \sum_{i=0}^{1+\log_2(N)} 2^i \times K = K \times (2^{\log_2(N)+2} - 1) = K \times (4 \times N - 1) = \Theta(N)$$

d)



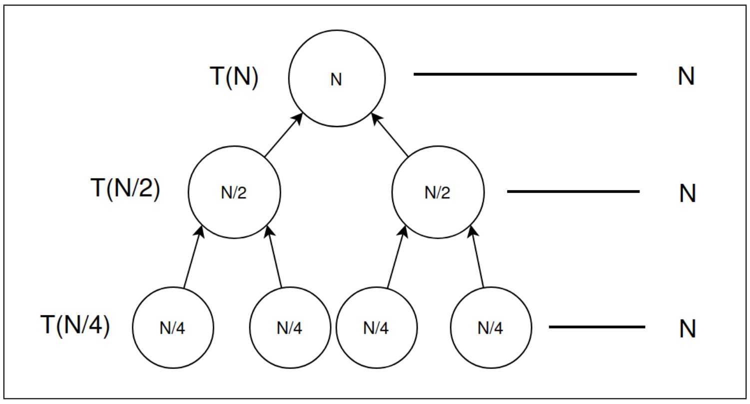
$$T(N) = K + \sum_{i=1}^{N} i = K + \frac{N \times (N+1)}{2} = \Theta(N^2)$$

e)



$$T(N) = K + \sum_{i=1}^{1+\log_2(N)} \frac{N}{2^i} = K + 2^{\log_2(N)+1} - 1 = K + 2 \times N - 1 = \Theta(N)$$

f)



$$T(N) = K + \sum_{i=1}^{1+\log_2(N)} N = K + N \times (1 + \log_2(N)) = \Theta(N \times \log_2(N))$$

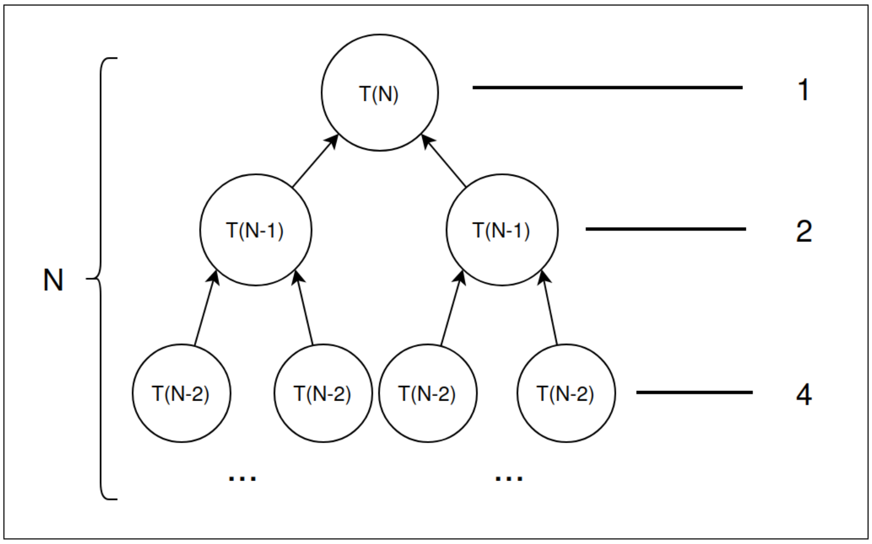
Exercício 2

$$T(N) = \begin{cases} 0 & N \leq 0 \\ 1 + N - 1 + T(N-1) & N > 0 \end{cases} = \begin{cases} 0 & N \leq 0 \\ N + T(N-1) & N > 0 \end{cases}$$

$$= \sum_{i=1}^{N} i = \frac{N \times (N+1)}{2} = \Theta(N^2)$$

Exercício 3

$$T(N) = \begin{cases} 0 & N \leq 0 \\ 1 + 2 \times T(N-1) & N > 0 \end{cases}$$



$$T(N) = \sum_{i=0}^{N-1} 2^i = 2^N - 1 = \Theta(2^N)$$

Exercício 4

$$T(N) = \begin{cases} 1 & N \leq 1 \\ T_{mergeH}(N) + 2 \times T(N/2) & N > 1 \end{cases} = \begin{cases} 1 & N \geq 1 \\ 2 \times N + 2 \times T(N/2) & N > 1 \end{cases}$$

$$= \sum_{i=1}^{1+\log_2(N)} 2 \times N = 2 \times N \times (1 + \log_2(N)) = \Theta(N \times \log_2(N))$$

Exercício 5

Árvores Equilibradas

$$T(N) = \begin{cases} 0 & N \leq 0 \\ 1 + 2 \times T(\frac{N-1}{2}) & N > 0 \end{cases} = \begin{cases} 0 & N \leq 0 \\ 1 + 2 \times T(N/2) & N > 0 \end{cases}$$

$$= \sum_{i=0}^{1+\log_2(N)} 2^i = 4 \times N - 1 = \Theta(N)$$

Árvores "Lista"

$$T(N) = \begin{cases} 0 & N \leq 0 \\ 1 + T(N-1) & N > 0 \end{cases} = \sum_{i=1}^{N} 1 = N = \Theta(N)$$

3 Análise de Caso Médio

Exercício 1

Pior caso: $T_{crescente}(N) = \sum_{i=1}^{N} 1 = N - 1 = O(N)$

Melhor caso: $T_{crescente}(N) = 1 = \Omega(1)$

$$T_{crescente}(N) = \Omega(1), O(N)$$

Caso médio da função `crescente()`:

$$\overline{T}_{crescente}(N) = (\sum_{i=1}^{N-1} (\frac{1}{2})^{i-1} \times (1 - \frac{1}{2}) \times i) + (\frac{1}{2})^{N-1} \times (N-1)$$

$$\overline{T}(N) = \sum_{i=1}^{N-1} (\frac{1}{2})^i \times i + (\frac{1}{2})^{N-1} \times (N-1) = \frac{1}{2} + \frac{1}{2} + \frac{3}{8} + \frac{1}{4} + \frac{5}{32} + ... < 2$$

Temos então $\overline{T}_{crescente}(N) = \Theta(1)$.

Caso médio da função `maxCresc()`:

$$\overline{T}_{maxCresc}(N) = \sum_{i=0}^{N-1} \overline{T}_{crescente}(N) = \sum_{i=0}^{N-1} 1 = \Theta(N)$$

Exercício 2

Exercício 3

Pior caso: $T_{strNdif}(N) = \sum_{i=1}^{N} 1 = N - 1 = O(N)$

Melhor caso: $T_{strNdif}(N) = 1 = \Omega(1)$

$$T_{strNdif}(N) = \Omega(1), O(N)$$

Caso médio da função `strNdif()`:

$$\overline{T}_{strNdif}(N) = \left(\sum_{i=0}^{N-1} \left(\frac{1}{26}\right)^i \times \left(1 - \frac{1}{26}\right) \times (i+1)\right) + \left(\frac{1}{26}\right)^N \times N$$

O somatório apresentado é composto pelos seguintes elementos:

- $\left(\frac{1}{26}\right)^i$, indica a probabilidade de i elementos serem iguais;
- $\left(1 - \frac{1}{26}\right)$, indica a probabilidade de duas letras serem diferentes;
- $(i+1)$, indica o custo de cada iteração.

Analisando o somatório verificamos que a probabilidade de as primeiras comparações serem falsas é maior, logo podemos concluir que o **caso médio** aproxima-se do **melhor caso** $\Omega(1)$.

Exercício 4

Para calcular o valor esperado do número de bit flips teremos que escrever a soma dos vários casos possíveis, pesados pela respectiva probabilidade. Para isso efectuamos uma contagem dos inputs correspondentes a cada um dos casos:

- metade dos bitvectors de comprimento N têm o bit menos significativo a 0;
- dos restantes, metade têm o **segundo** bit menos significativo a 0, i.e. terminam em 01;
- dos restantes, metade têm o **terceiro** bit menos significativo a 0, i.e. terminam em 011;
- e assim sucessivamente.

Temos então:

$$\overline{T}(N) = \frac{1}{2} \times 1flip + \frac{1}{4} \times 2flips + \frac{1}{8} \times 3flips + ... + \frac{1}{2^N} \times N + \frac{1}{2^N} \times N$$

Existem duas situações em que ocorrem N bit flips:

- todos os bits estão a 1, excepto o mais significativo (0111)
- todos os bits estão a 1 (1111)

$$\overline{T}(N) = \sum_{k=1}^{N} \frac{k}{2^k} + \frac{1}{2^N} \times N$$

Uma vez que $\sum_{k=1}^{\infty} \frac{k}{2^k} = 2$, temos $\overline{T}(N) < 2 = \Theta(1)$.

Exercício 5

Árvore equilibrada:

De forma a calcularmos o número médio de iterações do ciclo, concentremo-nos em primeiro lugar no caso em que esta procura termina com sucesso. O caso em que a procura termina em insucesso corresponde ao **pior caso**, que tem um custo **logarítmico**. Assumindo que o elemento existe com igual probabilidade em qualquer posição da árvore, as possíveis execuções desta função correspondem a:

- existe 1 possibilidade de o ciclo executar uma única vez: ou seja o custo será 1 com probabilidade $\frac{1}{N}$;
- existem 2 possibilidades de o ciclo executar duas vezes: ou seja o custo será 2 com probabilidade $\frac{2}{N}$;
- existem 3 possibilidades de o ciclo executar três vezes: ou seja o custo será 3 com probabilidade $\frac{3}{N}$;
- ...
- de uma forma genérica, existem 2^{k-1} possibilidades de o ciclo executar k vezes: ou seja o custo será de k com probabilidade $\frac{2^{k-1}}{N}$.

O custo esperado é então dado por:

$$
\begin{aligned}
\overline{T}(N) = & \sum_{k=1}^{\log_2 N-1} k \times \frac{2^{k-1}}{N} \\
= & \frac{1}{N} \times \sum_{k=1}^{\log_2 N-1} k \times 2^{k-1} \\
= & \frac{1}{N} \times ((\log_2(N) - 1) \times 2^{\log_2(N)} - (\log_2(N) \times 2^{\log_2(N)-1}) + 1) \\
= & \frac{1}{N} \times ((\log_2(N) - 1) \times N - \frac{1}{2} \times N \times \log_2(N) + 1) \\
= & \frac{1}{N} \times (\frac{1}{2} \times N \times \log_2(N) - N + 1) \\
= & \frac{1}{2} \times \log_2(N) - 1 + \frac{1}{N} \\
= & \Theta(\log_2 N)
\end{aligned}
$$

Verificamos então que o caso médio corresponde ao pior caso, pois a maioria dos elementos estão situados no último nível da árvore.

Árvore lista:

Nesta configuração de árvore, a procura binária passará a ser uma procura **sequencial**, e a altura da àrvore corresponderá ao seu número de elementos. Assumindo que o elemento existe com igual probabilidade em qualquer posição da árvore, as possíveis execuções desta função correspondem a:

- existe 1 possibilidade de o ciclo executar uma única vez: ou seja o custo será 1 com probabilidade $\frac{1}{N}$;

- existe 1 possibilidade de o ciclo executar duas vezes: ou seja o custo será 2 com probabilidade $\frac{1}{N}$;
- existe 1 possibilidade de o ciclo executar três vezes: ou seja o custo será 3 com probabilidade $\frac{1}{N}$;
- ...
- de uma forma genérica, existe 1 possibilidade de o ciclo executar k vezes: ou seja o custo será de k com probabilidade $\frac{1}{N}$.

O custo esperado é então dado por:

$$\begin{aligned}
\overline{T}(N) = & \sum_{k=0}^{N-1} k \times \frac{1}{N} \\
= & \frac{1}{N} \times \sum_{k=0}^{N-1} k \\
= & \frac{1}{N} \times \frac{N \times (N-1)}{2} \\
= & N - 1 \\
= & \Theta(N)
\end{aligned}$$

Concluimos que, se a árvore for degenerada, o custo médio será **linear**.

Exercício 6

Melhor caso: o array `a` não contém uma potência de 2, isto é, existe mais do que uma ocorrência do valor 1 no array, ou não existe nenhuma. O custo será $\Theta(N)$.

Pior caso: o array `a` contém uma potência de 2, isto é, existe apenas uma ocorrência do valor 1 no array. O custo será $\Theta(2^N)$.

4 Análise Amortizada

Exercício 1

Análise assimptótica da função `enqueue()`: a inserção de elementos é sempre feita na stack A, e as operações em stacks são constantes, a função tem sempre o mesmo comportamento seja qual for o input, logo:

$$T_{enqueue}(N) = \Theta(1)$$

Análise assimptótica da função `dequeue()`:

- **Melhor caso**: a stack B não está vazia, $T_{dequeue}(N) = \Omega(1)$.
- **Pior caso**: a stack B está vazia, então remove-se todos os elementos de A e coloca-se em B, $T_{dequeue}(N) = \sum_{i=1}^{N} 2 = 2 \times N = O(N)$.

$$T_{dequeue}(N) = \Omega(1), O(N)$$

Análise Amortizada:

A **função de potencial** deve traduzir o trabalho efetuado em cada estado, então iremos usar o número de elementos na stack A como medida:

$$\Phi(Q) = |A|$$

quantos mais elementos em A, maior será o trabalho futuro de os transferir para a stack B.

- $\Phi(Q) \geq 0$, o número de elementos na stack nunca é negativo
- $\Phi(Q_0) = 0$, ambas as stacks começam vazias

Função `enqueue()`:

O custo real de adicionar um elemento à stack A é constante, indicado no enunciado, logo $c_i = 1$. O potencial no estado i é $|A|+1$, pois a stack A tem mais um elemento, enquanto que $\Phi_{i-1} = |A|$.

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$$
$$\hat{c}_i = 1 + |A| + 1 - |A| = 2$$

Logo, podemos concluir que o custo amortizado de cada execução da função `enqueue()` é **constante**.

Função `dequeue()`:

Existem duas situações a analisar nesta função, quando a stack B não está vazia e quando está vazia. Quando a stack B não está vazia, a análise é simples, não há alteração no número de elementos da stack A, por isso a variação de potencial é nula, $\Delta\Phi = 0$, então o custo amortizado será igual ao custo real, que é 1.

Quando a stack B está vazia, todos os elementos de A são passados para B. o custo desta operação é $2|A| + 1$, dado que é feito um `pop` e um `push` a todos os elementos, e por último um `pop`. O potencial no estado atual é 0, pois a stack A foi esvaziada, e no estado anterior é $|A|$.

$$\hat{c}_i = c_i + \Phi_i + \Phi_{i-1}$$
$$\hat{c}_i = 2|A| + 1 + 0 - |A|$$
$$\hat{c}_i = |A| + 1$$

O valor obtido indica que o custo amortizado de `dequeue()` é linear, mas estamos apenas a considerar uma sequência de operações `dequeue()`, e não as operações `enqueue()` que colocaram os elementos na stack A. Tendo isto em conta o custo amortizado é constante, pois o custo de remover todos os elementos de A é amortizado pelo custo de os colocar lá, ou seja, numa única operação o custo é linear, mas foi necessário N iterações da função `enqueue()`, com um custo constante, para amortizar o custo de `dequeue()`.

Exercício 2

Seja N o número de elementos na stack, a função de potencial a utilizar será $\Phi(Q) = N$. Existem duas situações a analisar: (1) k é menor que o número de elementos na stack, logo o custo real será k; (2) o número de elementos na stack é menor que k, logo o custo real será N.

(1) No caso de $k < N$, o custo amortizado será:

$$\hat{c}_i = k + N - k - N = 0$$

O potencial no estado anterior é N, que corresponde ao número de elementos da stack, o potencial no estado atual é $N - k$, pois foram retirados k elementos à stack.

(2) No de caso de $k \geq N$, o custo amortizado será:

$$\hat{c}_i = N + 0 - N = 0$$

Neste caso, foram removidos todos os elementos da stack, daí o potencial no estado atual ser 0, pois a stack está vazia. Podemos então concluir que o **custo amortizado** da função `multiPop()` é O(1).

Exercício 3

Análise **assimptótica** da função `insert_rem`:

- **Melhor caso**: inserir `x` à cabeça da lista, $T_{insert\_rem}(N) = \Omega(1)$.
- **Pior caso**: inserir `x` no fim da lista, $T_{insert\_rem}(N) = \sum_{i=1}^{N} 1 = O(N)$.

$$T_{insert\_rem}(N) = \Omega(1), O(N)$$

Análise **agregada**:

i	input	output	c_i
1		20	1
2	20	70	2
3	70	$60 \rightarrow 70$	1
4	$60 \rightarrow 70$	$30 \rightarrow 60 \rightarrow 70$	1
5	$30 \rightarrow 60 \rightarrow 70$	$40 \rightarrow 60 \rightarrow 70$	2
6	$40 \rightarrow 60 \rightarrow 70$	$50 \rightarrow 60 \rightarrow 70$	2
7	$50 \rightarrow 60 \rightarrow 70$	$10 \rightarrow 50 \rightarrow 60 \rightarrow 70$	1
8	$10 \rightarrow 50 \rightarrow 60 \rightarrow 70$	80	5
...			

Custo das 8 operações:

$$1 + 2 + 1 + 1 + 2 + 2 + 1 + 5 = 15$$

Custo **amortizado** por operação:

$$\frac{15}{8} = 1.875$$

Seja N o número de elementos na lista, a função de **potencial** será a seguinte:

$$\Phi(S) = N$$

O custo de executar a função `insert_rem()` é $k + 1$, sendo k o número de elementos removidos e 1 o custo de adicionar. Podemos então calcular o custo amortizado:

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
\hat{c}_i &= k + 1 + (N - (k + 1)) - N \\
\hat{c}_i &= 2
\end{aligned}$$

Logo, podemos concluir que o custo amortizado de `insert_rem()` é **constante**.