Lista 1 - MC458 - Gabarito
Tema: Divisão e Conquista

1. **Busca Binária**

   a) **Pseudocódigo:**

```
BUSCA_BINARIA(A, i, j, x)
    se i > j
        retornar -1
    m <- (i + j) / 2
    se A[m] == x
        retornar m
    se A[m] > x
        retornar BUSCA_BINARIA(A, i, m - 1, x)
    senão
        retornar BUSCA_BINARIA(A, m + 1, j, x)
```

   b) **Complexidade:**
   Seja $n = j - i + 1$. A cada iteração $n \leftarrow \frac{n}{2}$.

$$\frac{n}{2^k} \leq 1 \quad \text{(seja } k = \text{n}^\circ \text{ de operações no pior caso)}$$

$$2^k \geq n$$

$$\log_2 2^k \geq \log_2 n$$

$$k \geq \log_2 n$$

$$\text{logo a complexidade é } \Theta(\log n)$$

2. **Merge Sort**

Algoritmo 1: MergeSort
Entrada: Um arranjo com n
Saída: Um arranjo com os mesmos números ordenados

1 **se** $n \geq 2$ **então**
2 A = Recursivamente ordenar a primeira metade do arranjo de entrada;
3 B = Recursivamente ordenar a segunda metade do arranjo de entrada;
4 C = Intercalar (Merge) as duas partes ordenadas A e B em uma;
5 **devolva** C;

Algoritmo 2: Merge
Entrada: A e B arranjos ordenados com $m/2$
Saída: Arranjo C de tamanho m com os mesmos elementos de A e B mas ordenados

```
1  i = 1;
2  j = 1;
3  para k de 1 até m faça
4     se A[i] < B[j] então
5        C[k] = A[i];
6        i++;
7     senão
8        C[k] = B[j];
9        j++;
10 devolva C;
```

a) **Prova de corretude:**

obs: Talvez esse item seja mais interessante ao avançar um pouco mais no conteúdo, caso ainda não esteja familiarizado com o conceito de invariante de laço, não se preocupe, deveria ter colocado essa questão em listas futuras (PAD).

Uma **invariante de laço** é uma propriedade utilizada para provar corretudes de laços, a qual está associada a determinada posição de um laço e é satisfeita toda execução do laço. Sendo assim, utilizaremos esse conceito para provar por indução a corretude do algoritmo **Merge.**

I) Provando a função `Merge`:

Invariante de laço: No início da iteração k (linha 3), o subarranjo $C[1 \dots k-1]$ contém os $k-1$ menores elementos combinados de A e B de forma ordenada. Além disso, $A[i]$ e $B[j]$ são os menores elementos de A e B que ainda não foram copiados para C.

Caso Base ($k = 1$): O arranjo $C[1 \dots 0]$ está vazio, cumprindo a condição de estar ordenado e conter zero elementos. $A[1]$ e $B[1]$ são os menores de seus respectivos arranjos (que já são recebidos ordenados).

Passo Indutivo: Assuma que a invariante vale no início da iteração k. Na linha 4, comparamos $A[i]$ e $B[j]$. Como ambos são os menores disponíveis de suas respectivas metades, o menor entre eles é garantidamente o menor elemento global restante.

- Se $A[i] < B[j]$, $A[i]$ é inserido em $C[k]$ e i é incrementado. $C[1 \dots k]$ agora contém os k menores elementos ordenados.
- Se $B[j] \leq A[i]$, $B[j]$ é inserido e j é incrementado, mantendo a mesma lógica.

Ao final da iteração, a invariante é preservada para o passo $k+1$.

Término: O laço encerra quando $k = m+1$. O arranjo C conterá todos os m elementos de A e B ordenados.

II) Provando a função `MergeSort`:

Por indução forte sobre o tamanho n do arranjo. Queremos provar que o algoritmo ordena qualquer vetor de tamanho n.

Caso Base ($n = 1$): A condição $n \geq 2$ falha. Um arranjo de um único elemento já está trivialmente ordenado.

Passo Indutivo: Assuma que o algoritmo ordena perfeitamente qualquer arranjo de tamanho $< n$. Para um arranjo de tamanho $n \geq 2$, o algoritmo o divide em duas partes e as processa recursivamente. Pela hipótese indutiva, as chamadas devolvem A e B totalmente ordenados. Em seguida, `Merge` é invocado. Como provado na etapa anterior que `Merge` funde dois arranjos ordenados em um único ordenado, o arranjo resultante C estará correto.

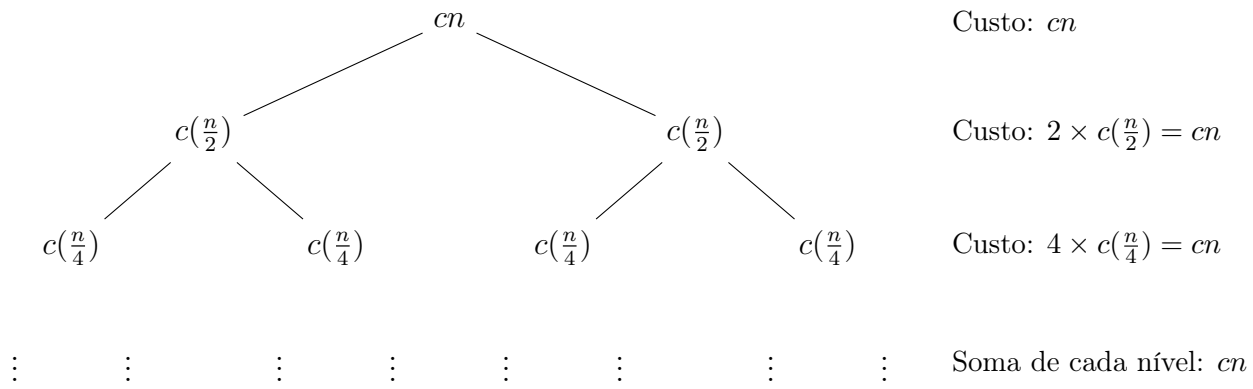
b) **Complexidade e Árvore de Recursão:**
Para determinarmos a equação de recorrência $T(n)$, que representa o tempo de execução do `MergeSort` para um arranjo de tamanho n, analisamos os custos individuais do Algoritmo 1:

- **Divisão:** O algoritmo divide o problema conceitualmente ao meio. O custo para calcular o meio (ou lidar com os índices) é constante, ou seja, $O(1)$.
- **Conquista (Chamadas Recursivas):** O algoritmo realiza duas chamadas recursivas (linhas 2 e 3) para ordenar as metades. Como cada metade tem tamanho $n/2$, o custo exato dessas chamadas é $2T(n/2)$.
- **Combinação (Merge):** A função `Merge` (Algoritmo 2) itera sobre os elementos para fundi-los. O laço `para` da linha 3 executa m vezes (onde $m = n$ na junção completa). Como as operações internas (comparações e atribuições) levam tempo constante c, o custo total de intercalação é linear: cn.

Somando essas etapas, chegamos à equação de recorrência formal:

$$T(n) = 2T(n/2) + cn$$

Abaixo, a representação visual da árvore de expansão da recorrência detalhando o custo por nível:



A árvore continua a se ramificar dividindo o tamanho do arranjo pela metade até atingir o caso base, onde o subproblema tem tamanho 1. A profundidade máxima da árvore é alcançada quando:

$$\frac{n}{2^h} = 1 \implies h = \log_2 n$$

A árvore possui altura $\log_2 n$. Como cada nível realiza um trabalho total exatamente igual a cn, e existem $\log_2 n + 1$ níveis no total (contabilizando o nível 0 da raiz), o custo total de execução é a soma de todos os níveis:

$$T(n) = \sum_{i=0}^{\log_2 n} cn = cn(\log_2 n + 1) = cn \log_2 n + cn$$

Descartando as constantes e os termos de menor grau para a análise assintótica, concluímos que o tempo de execução é $O(n \log n)$.

3. **Soma do Subarranjo Máximo (Maximum Subarray Sum)**
O Problema: Dado um arranjo de n números inteiros, encontre o subarranjo contíguo cuja soma dos elementos seja máxima.

a) **Ideia e Pseudocódigo:**
Ideia geral: Pela estratégia de divisão e conquista, dividimos o arranjo ao meio. O subarranjo de soma máxima só pode estar em um de três lugares: totalmente na metade esquerda, totalmente na metade direita, ou cruzando o ponto médio. Resolvemos recursivamente para as metades e, para o cruzamento, expandimos linearmente a partir do centro. O resultado será o máximo entre essas três opções.

Algoritmo 1: `MaxCruzamento`
Entrada: Arranjo A, índices `baixo`, `meio` e `alto`

```
1 soma_esq = −∞; soma_atual = 0;
2 para i de meio descendo até baixo faça
3     soma_atual = soma_atual + A[i];
4     se soma_atual > soma_esq então soma_esq = soma_atual;
5 soma_dir = −∞; soma_atual = 0;
6 para j de meio + 1 até alto faça
7     soma_atual = soma_atual + A[j];
8     se soma_atual > soma_dir então soma_dir = soma_atual;
9 devolva (soma_esq + soma_dir);
```

Algoritmo 2: `SubarranjoMaximo`
Entrada: Arranjo A, índices `baixo` e `alto`

```
1 se baixo == alto então
2     devolva A[baixo];
3 meio = ⌊(baixo + alto)/2⌋;
4 esq = SubarranjoMaximo(A, baixo, meio);
5 dir = SubarranjoMaximo(A, meio + 1, alto);
6 cruz = MaxCruzamento(A, baixo, meio, alto);
7 devolva max(esq, dir, cruz);
```

b) **Prova de corretude:**
Utilizaremos indução forte sobre o tamanho do arranjo n para provar que o algoritmo é capaz de encontrar a soma do subarranjo máximo.

Caso base ($n = 1$): A linha 1 do Algoritmo 2 é acionada (quando `baixo == alto`) e retorna o único elemento. Como o único subarranjo contíguo possível é o próprio elemento, a sua soma é o valor máximo. A base é válida.

Passo indutivo: Assuma que o algoritmo resolve corretamente o problema para qualquer arranjo de tamanho $k < n$. Para um arranjo de tamanho $n > 1$, o subarranjo contíguo de soma máxima deve pertencer a exatamente um dos três casos:

i. Estar inteiramente contido na metade esquerda (índices $\leq$ `meio`).
ii. Estar inteiramente contido na metade direita (índices $>$ `meio`).
iii. Cruzar o ponto médio (inicia na metade esquerda e termina na metade direita).

Analisando os casos 1 e 2: A divisão na linha 3 garante que o problema foi quebrado em subarranjos estritamente menores. Pela hipótese indutiva, as chamadas recursivas nas linhas 4 (`esq`) e 5 (`dir`) retornam corretamente a soma máxima para os cenários onde o subarranjo não cruza a fronteira.

Analisando o caso 3 (Corretude da função `MaxCruzamento`): Qualquer subarranjo contíguo que cruza o ponto médio **obrigatoriamente** precisa incluir os elementos vizinhos $A[\texttt{meio}]$ e $A[\texttt{meio} + 1]$. Podemos quebrar esse subarranjo em duas partes: um sufixo da metade esquerda $A[i \dots \texttt{meio}]$ e um prefixo da metade direita $A[\texttt{meio} + 1 \dots j]$.

No Algoritmo 1, o primeiro laço (linhas 2-4) fixa uma "âncora"no índice `meio` e itera para a esquerda até o índice `baixo`. Ele testa todos os comprimentos possíveis de forma exaustiva e salva o maior resultado em `soma_esq`. O segundo laço (linhas 6-8) faz o processo simétrico: fixa uma âncora em `meio`+1 e cresce iterativamente para a direita até o índice `alto`, salvando o máximo em `soma_dir`.

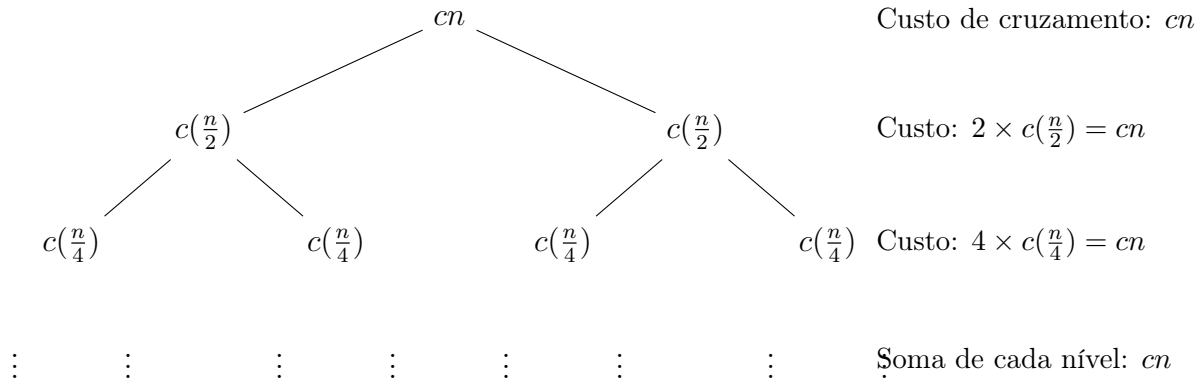
Como as duas metades procuradas são adjacentes e independentes (a escolha de i não afeta a escolha de j), a soma `soma_esq` + `soma_dir` (linha 9) corresponde matematicamente ao subarranjo ótimo que passa pela fronteira.

Conclusão: Como o algoritmo testa corretamente todas as três únicas localizações possíveis para a resposta e a linha 7 retorna o valor máximo (`max`) entre as três, fica provado exaustivamente que o algoritmo sempre retornará a soma do subarranjo máximo global.

c) **Complexidade e Árvore de Recursão:**
A divisão calcula o ponto médio em $O(1)$. A função `SubarranjoMaximo` realiza duas chamadas recursivas de tamanho $n/2$, custando $2T(n/2)$. A função `MaxCruzamento` itera do centro para as bordas cobrindo os n elementos no total, custando cn (tempo linear). A relação de recorrência é idêntica à do Merge Sort:

$$T(n) = 2T(n/2) + cn$$

Abaixo, a representação da árvore de recursão para este custo:



A árvore de recursão divide o problema por 2 a cada nível, atingindo o caso base ($n/2^h = 1$) na profundidade $h = \log_2 n$. Como temos $\log_2 n + 1$ níveis (contando a raiz) e o trabalho total executado por todos os nós em qualquer nível é sempre cn, o custo somado é:

$$T(n) = \sum_{i=0}^{\log_2 n} cn = cn(\log_2 n + 1) \implies O(n \log n)$$

4. **Exponenciação Rápida**
O Problema: Calcular x^n (onde x é um número real e n é um inteiro não negativo) de forma eficiente.

a) **Pseudocódigo:**

Algoritmo 3: `ExpRapida`
Entrada: Base x (real), expoente n (inteiro não negativo)
Saída: x^n calculado de forma eficiente

```
1 se n == 0 então
2    devolva 1;
3 se n mod 2 == 0 então    // Se o expoente for par
4    y = ExpRapida(x, n/2);
5    devolva y × y;
6 senão    // Se o expoente for ímpar
7    y = ExpRapida(x, (n − 1)/2);
8    devolva x × y × y;
```

b) **Complexidade e Árvore de Recursão:**
Diferente dos algoritmos anteriores, este realiza **apenas uma** chamada recursiva por nível, dividindo o tamanho do expoente n pela metade. As operações locais em cada chamada (verificar paridade, divisões por 2 e multiplicações) ocorrem em tempo constante, que denotaremos por c.

A equação de recorrência formaliza-se como:

$$T(n) = T(n/2) + c$$

Desenhando a árvore de recursão, percebemos que ela não se ramifica, degenerando em um único caminho linear do topo até a base:

c — Subproblema de tamanho $n \rightarrow$ Custo: c

$\Big|$

c — Subproblema de tamanho $n/2 \rightarrow$ Custo: c

$\Big|$

c — Subproblema de tamanho $n/4 \rightarrow$ Custo: c

$\vdots$ — Caso base de tamanho $1 \rightarrow$ Custo: c

A árvore continua sendo dividida por 2 até atingir o caso base (onde $n = 0$ ou $n = 1$). A altura máxima h da árvore é encontrada pela relação $\frac{n}{2^h} = 1 \implies h = \log_2 n$. Como a árvore possui $\log_2 n + 1$ níveis no total (incluindo o nível 0) e o custo de trabalho em **cada nível inteiro** é apenas a constante c, o custo total de execução é a soma de c ao longo de toda a altura da árvore:

$$T(n) = \sum_{i=0}^{\log_2 n} c = c(\log_2 n + 1) = c \log_2 n + c$$

Descartando as constantes para a análise assintótica, demonstramos que a complexidade de tempo é estritamente $O(\log n)$.

5. **Encontrar um Pico (Peak Finding 1D)**
O Problema: Dado um arranjo $A[1..n]$ de números inteiros onde elementos adjacentes são estritamente diferentes ($A[i] \neq A[i+1]$), dizemos que um elemento $A[i]$ é um "pico" se ele for estritamente maior que seus vizinhos imediatos. Para as bordas, $A[1]$ é pico se $A[1] > A[2]$, e $A[n]$ é pico se $A[n] > A[n-1]$. O vetor não está ordenado e pode haver múltiplos picos. O objetivo é encontrar **qualquer um** dos picos acessando o menor número possível de elementos.

   a) **Ideia e Pseudocódigo:**
   Ideia geral: A estratégia é similar à da Busca Binária. Avaliamos o elemento central do subarranjo. Se ele não for um pico, obrigatoriamente um de seus vizinhos é maior, formando uma "subida"(encosta). Como o vetor é finito, se continuarmos descartando a metade "menor"e seguindo na direção da subida, garantidamente encontraremos um pico (seja no meio do caminho ou em uma das bordas do arranjo).

Algoritmo 4: `EncontrarPico`
Entrada: Arranjo A, índices `baixo` e `alto`
Saída: Índice de um elemento que é um pico local

```
1 m = ⌊(baixo + alto)/2⌋;
2 se (m > baixo) E (A[m − 1] > A[m]) então
3     devolva EncontrarPico(A, baixo, m - 1);
4 senão se (m < alto) E (A[m + 1] > A[m]) então
5     devolva EncontrarPico(A, m + 1, alto);
6 senão
7     devolva m;
```

   b) **Complexidade e Árvore de Recursão:**
   Em cada chamada da função `EncontrarPico`, calculamos o meio e realizamos um

número constante de verificações lógicas e acessos ao arranjo. Isso custa $O(1)$, que chamaremos de c. Independentemente de qual condição seja satisfeita no `se/senão`, o algoritmo faz, no máximo, **uma** única chamada recursiva para a metade do arranjo atual.

Portanto, a equação de recorrência é idêntica à da Exponenciação Rápida:

$$T(n) = T(n/2) + c$$

A representação em formato de árvore também é linear, sem ramificações horizontais, pois resolvemos apenas um subproblema por vez:

c — Tamanho $n \to$ Custo na ativação: c

$\vert$

c — Tamanho $n/2 \to$ Custo na ativação: c

$\vert$

c — Tamanho $n/4 \to$ Custo na ativação: c

$\vdots$ — Tamanho 1 (Caso Base) $\to$ Custo: c

A árvore continua sendo dividida até encontrar o caso base onde há apenas um elemento ($n/2^h = 1$), o que significa que a profundidade da árvore é $h = \log_2 n$.

Como o custo em cada nível (que possui apenas um nó) é estritamente a constante c, e existem $\log_2 n + 1$ níveis, a soma total do tempo de execução é:

$$T(n) = \sum_{i=0}^{\log_2 n} c = c(\log_2 n + 1) = c \log_2 n + c$$

Ignorando as constantes aditivas e multiplicativas, provamos que a complexidade de tempo de execução deste algoritmo é $O(\log n)$.