BCC202 - Estruturas de Dados I

Aula 05: Análise de Algoritmos (Parte 1)

Pedro Silva

Universidade Federal de Ouro Preto, UFOP
Departamento de Computação, DECOM
Email: **silvap@ufop.edu.br**

Conteúdo

Conteúdo

Análise de Algoritmos

- **Analisar** um algoritmo consiste em "**verificar**" o **custo** do algoritmo em relação ao:
  - **Tempo** gasto para executá-lo.
  - **Espaço** (**memória**) ocupado em sua execução.

- Esta análise é necessária para que se possa escolher o algoritmo mais adequado para resolver um dado problema.
- É essencialmente importante em áreas de pesquisa operacional, otimização, teoria dos grafos, estatística, probabilidades, entre outras.

Cálculo do custo real pela execução do algoritmo

- Medidas são **inadequadas** e o resultado não pode ser generalizado.
- Tais medidas são dependentes do compilador, que pode favorecer algumas construções em detrimento de outras.
- Resultados dependem do *hardware*.
- Quando grandes quantidades de memória são utilizadas, as medidas de tempo podem depender deste aspecto.

Cálculo do custo real pela execução do algoritmo

- Apesar disto, há argumentos a favor de se obter medidas reais da execução do algoritmo.
- **Um exemplo**: quando há vários algoritmos distintos para resolver um mesmo problema, todos possuindo um custo de execução dentro de uma mesma **ordem de grandeza**.

Conteúdo

Possibilidades de análise

- Um algoritmo em particular.
- Uma classe de algoritmos.

Possibilidades de análise

- Um algoritmo em particular.
- Uma classe de algoritmos.

Análise de um algoritmo em particular

- Qual é o custo de usar um dado algoritmo para resolver um problema específico?
- Características que devem ser investigadas:
  - Análise do número de vezes que cada parte do algoritmo deve ser executada (**tempo**).
  - Estudo da quantidade de memória necessária (**espaço**).

Análise de uma classe de algoritmos

- Qual é o algoritmo de menor custo possível para resolver um problema em particular?
- Toda uma **família** de algoritmos é investigada.
- Procura-se identificar um que seja o melhor possível.
- Coloca-se **limites** para a complexidade computacional dos algoritmos pertencentes à classe.

Custo de um algoritmo

- O menor custo possível para resolver problemas de uma classe nos dá a dificuldade inerente para resolver o problema.

- **Quando o custo de um algoritmo é igual ao menor custo possível, o algoritmo é ótimo para a medida de custo considerada**.

- Podem existir vários algoritmos ótimos para resolver o mesmo problema.

- Se a mesma medida de custo é aplicada a diferentes algoritmos, então é possível compará-los e escolher o mais adequado.

Custo de um algoritmo

- Utilizaremos um modelo matemático baseado em um computador idealizado.
- Deve ser especificado o conjunto de operações e seus custos de execução.
- É mais usual ignorar o custo de algumas operações e considerar apenas as mais significativas.
- Ex.: Em **algoritmos de ordenação** consideramos o número de comparações entre os elementos do conjunto a ser ordenado e ignoramos as demais operações.

Função de complexidade

- Para medir o custo de execução de um algoritmo definiremos uma função de complexidade ou função de custo f.
- Função de **complexidade de tempo**: $f(n)$ mede o tempo necessário para executar um algoritmo em um problema de tamanho n.
- Função de **complexidade de espaço**: $f'(n)$ mede a memória necessária para executar um algoritmo em um problema de tamanho n.

Função de complexidade

- Utilizaremos f para denotar uma função de complexidade de **tempo** daqui para frente.
- A complexidadede tempo na realidade **não** representa tempo diretamente:
  - Representa o número de vezes que determinadas operações, ditas **relevantes**, são executadas.

Exemplo: Maior elemento de um vetor

- Considere o algoritmo para encontrar o maior elemento de um vetor de inteiros $A[n]$; $n >= 1$.

```
int Max(int* A, int n) {
  int i, Temp;
  Temp = A[0];
  for(i = 1; i < n; i++)
    if(Temp < A[i]) // Comparação envolvendo os elementos
      Temp = A[i];
  return Temp;
}
```

- Seja f uma função de complexidade tal que $f(n)$ é o número de comparações envolvendo os elementos de A, se A contiver n elementos.
- Qual é a função $f(n)$?

Exemplo: Maior elemento de um vetor

- Considere o algoritmo para encontrar o maior elemento de um vetor de inteiros $A[n]$; $n >= 1$.

```
int Max(int* A, int n) {
  int i, Temp;
  Temp = A[0];
  for(i = 1; i < n; i++)
    if(Temp < A[i]) // Comparação envolvendo os elementos
      Temp = A[i];
  return Temp;
}
```

- Seja f uma função de complexidade tal que $f(n)$ é o número de comparações envolvendo os elementos de A, se A contiver n elementos.
- Qual é a função $f(n)$? $f(n) = n - 1$.

Exemplo: Maior elemento de um vetor

- **Teorema**: Qualquer algoritmo para encontrar o maior elemento de um conjunto com n elementos, $n >= 1$, faz pelo menos $n - 1$ comparações.
- **Prova**: Cada um dos $n - 1$ elementos tem de ser investigado por meio de comparações, que é menor do que algum outro elemento.
  - Logo, $n - 1$ comparações são necessárias.

O teorema acima nos diz que, se o número de comparações for utilizado como medida de custo, então a função $\boldsymbol{Max}$ do programa anterior é ótima.

Tamanho da entrada de dados

- A medida do custo de execução de um algoritmo **depende principalmente** do **tamanho da entrada** dos dados.
- É comum considerar o **tempo de execução** de um programa como uma **função do tamanho da entrada**.
- Para alguns algoritmos, o custo de execução é uma função da entrada particular dos dados, não apenas do tamanho da entrada.
  - Ou seja, o custo pode ser diferente para entradas distintas, mas de mesmo tamanho.

Tamanho da entrada de dados

- A medida do custo de execução de um algoritmo **depende principalmente** do **tamanho da entrada** dos dados.
- É comum considerar o **tempo de execução** de um programa como uma **função do tamanho da entrada**.
- Para alguns algoritmos, o custo de execução é uma função da entrada particular dos dados, não apenas do tamanho da entrada.
  - Ou seja, o custo pode ser diferente para entradas distintas, mas de mesmo tamanho.

Tamanho da entrada de dados

- A medida do custo de execução de um algoritmo **depende principalmente** do **tamanho da entrada** dos dados.
- É comum considerar o **tempo de execução** de um programa como uma **função do tamanho da entrada**.
- Para alguns algoritmos, o custo de execução é uma função da entrada particular dos dados, não apenas do tamanho da entrada.
  - Ou seja, o custo pode ser diferente para entradas distintas, mas de mesmo tamanho.

Tamanho da entrada de dados

- No caso da função Max do programa do exemplo, o custo é uniforme sobre todos os problemas de tamanho n.
- Já para um algoritmo de ordenação isso não ocorre: se os dados de entrada já estiverem quase ordenados, então pode ser que o algoritmo trabalhe menos.

Definição

- **Melhor caso**: menor tempo de execução sobre todas as entradas de tamanho n.
- **Pior caso**: maior tempo de execução sobre todas as entradas de tamanho n.
- **Caso médio** (ou *caso esperado*): média dos tempos de execução de todas as entradas de tamanho n.

Melhor caso <= Caso médio <= Pior caso

Análise do caso médio

- A análise do **caso médio** é geralmente muito mais difícil de obter do que as análises do melhor e do pior casos.
- Supõe-se uma **distribuição de probabilidades** sobre o conjunto de entradas de tamanho n.
- É comum supor uma distribuição de probabilidades em que todas as entradas possíveis são **igualmente prováveis**.
  - Na prática isso nem sempre é verdade.

Análise do caso médio

- A análise do **caso médio** é geralmente muito mais difícil de obter do que as análises do melhor e do pior casos.
- Supõe-se uma **distribuição de probabilidades** sobre o conjunto de entradas de tamanho n.
- É comum supor uma distribuição de probabilidades em que todas as entradas possíveis são **igualmente prováveis**.
  - Na prática isso nem sempre é verdade.

Análise do caso médio

- A análise do **caso médio** é geralmente muito mais difícil de obter do que as análises do melhor e do pior casos.
- Supõe-se uma **distribuição de probabilidades** sobre o conjunto de entradas de tamanho $\boldsymbol{n}$.
- É comum supor uma distribuição de probabilidades em que todas as entradas possíveis são **igualmente prováveis**.
  - Na prática isso nem sempre é verdade.

Exemplo: Registros de um arquivo

- Considere o problema de acessar os **registros** de um arquivo.
- Cada registro contém uma **chave** única que é utilizada para recupera-lo.
- **O problema**: dada uma chave qualquer, localize o registro que contenha esta chave.
- O **algoritmo de pesquisa** mais simples é o que faz a **pesquisa sequencial**.

Exemplo: Registros de um arquivo

- Considere o problema de acessar os **registros** de um arquivo.
- Cada registro contém uma **chave** única que é utilizada para recupera-lo.
- **O problema**: dada uma chave qualquer, localize o registro que contenha esta chave.
- O **algoritmo de pesquisa** mais simples é o que faz a **pesquisa sequencial**.

Exemplo: Registros de um arquivo

- Considere o problema de acessar os **registros** de um arquivo.
- Cada registro contém uma **chave** única que é utilizada para recupera-lo.
- **O problema**: dada uma chave qualquer, localize o registro que contenha esta chave.
- O **algoritmo de pesquisa** mais simples é o que faz a **pesquisa sequencial**.

Exemplo: Registros de um arquivo

- Considere o problema de acessar os **registros** de um arquivo.
- Cada registro contém uma **chave** única que é utilizada para recupera-lo.
- **O problema**: dada uma chave qualquer, localize o registro que contenha esta chave.
- O **algoritmo de pesquisa** mais simples é o que faz a **pesquisa sequencial**.

Exemplo: Registros de um arquivo (Análise)

- Seja $f(n)$ uma função de complexidade, onde n corresponde ao número de registros.
- A complexidade será definida pelo número de registros consultados (número de comparações de chaves):
  - **Melhor caso:** O registro procurado é o **primeiro** consultado!!!
    - $f(n) = 1$.
  - **Pior caso:** O registro procurado é o **último** consultado!!!
    - $f(n) = n$.
  - **Caso médio:**
    - Nem sempre é tão simples de se calcular.
    - Como fazer para este problema?

Exemplo: Registros de um arquivo (Análise)

- Seja $f(n)$ uma função de complexidade, onde n corresponde ao número de registros.
- A complexidade será definida pelo número de registros consultados (número de comparações de chaves):
  - **Melhor caso:** O registro procurado é o **primeiro** consultado!!!
    - $f(n) = 1$.
  - **Pior caso:** O registro procurado é o **último** consultado!!!
    - $f(n) = n$.
  - **Caso médio:**
    - Nem sempre é tão simples de se calcular.
    - Como fazer para este problema?

Exemplo: Registros de um arquivo (Análise)

- Seja $f(n)$ uma função de complexidade, onde n corresponde ao número de registros.
- A complexidade será definida pelo número de registros consultados (número de comparações de chaves):
  - **Melhor caso:** O registro procurado é o **primeiro** consultado!!!
    - $f(n) = 1$.
  - **Pior caso:** O registro procurado é o **último** consultado!!!
    - $f(n) = n$.
  - **Caso médio:**
    - Nem sempre é tão simples de se calcular.
    - Como fazer para este problema?

Exemplo: Registros de um arquivo (Análise)

- Seja $f(n)$ uma função de complexidade, onde n corresponde ao número de registros.
- A complexidade será definida pelo número de registros consultados (número de comparações de chaves):
  - **Melhor caso:** O registro procurado é o **primeiro** consultado!!!
    - $f(n) = 1$.
  - **Pior caso:** O registro procurado é o **último** consultado!!!
    - $f(n) = n$.
  - **Caso médio:**
    - Nem sempre é tão simples de se calcular.
    - **Como fazer para este problema?**

Exemplo: Registros de um arquivo (Análise)

- Seja $f(n)$ uma função de complexidade, onde n corresponde ao número de registros.
- A complexidade será definida pelo número de registros consultados (número de comparações de chaves):
  - **Melhor caso:** O registro procurado é o **primeiro** consultado!!!
    - $f(n) = 1$.
  - **Pior caso:** O registro procurado é o **último** consultado!!!
    - $f(n) = n$.
  - **Caso médio:**
    - Nem sempre é tão simples de se calcular.
    - **Como fazer para este problema?**

Exemplo: Registros de um arquivo (Análise do Caso médio)

- No estudo do caso médio, vamos considerar que toda pesquisa recupera um registro.
- Se $\boldsymbol{p_i}$ for a probabilidade de que o *i-ésimo* registro seja procurado, e considerando que para recupera-lo são necessárias $\boldsymbol{i}$ comparações, então:

$$f(n) = 1 * p_1 + 2 * p_2 + 3 * p_3 + ... + n * p_n$$

Exemplo: Registros de um arquivo (Análise do Caso médio)

- Para calcular $f(n)$ basta conhecer a distribuição de probabilidades p_i.
- Se cada registro tiver a mesma probabilidade de ser acessado que todos os outros, então:

$$p_i = \frac{1}{n}, \text{para } 1 <= i <= n$$

- Logo:

$$f(n) = \frac{1}{n}(1 + 2 + 3 + ... + n) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}$$

Exemplo: Registros de um arquivo (Análise do Caso médio)

- Para calcular $\boldsymbol{f(n)}$ basta conhecer a distribuição de probabilidades $\boldsymbol{p_i}$.
- Se cada registro tiver a mesma probabilidade de ser acessado que todos os outros, então:

$$p_i = \frac{1}{n}, \text{para } 1 <= i <= n$$

- Logo:

$$f(n) = \frac{1}{n}(1 + 2 + 3 + \ldots + n) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}$$

Exemplo: Registros de um arquivo (Análise do Caso médio)

- Para calcular $\boldsymbol{f(n)}$ basta conhecer a distribuição de probabilidades $\boldsymbol{p_i}$.
- Se cada registro tiver a mesma probabilidade de ser acessado que todos os outros, então:

$$p_i = \frac{1}{n}, \text{para } 1 <= i <= n$$

- Logo:

$$f(n) = \frac{1}{n}(1 + 2 + 3 + ... + n) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}$$

Exemplo: Registros de um arquivo (Análise)

- **Melhor caso:**

Exemplo: Registros de um arquivo (Análise)

- **Melhor caso:**
  - O registro procurado é o **primeiro** consultado!!!
  - $f(n) = 1$.

Exemplo: Registros de um arquivo (Análise)

- **Melhor caso:**
  - O registro procurado é o **primeiro** consultado!!!
  - $f(n) = 1$.
- **Pior caso:**

Exemplo: Registros de um arquivo (Análise)

- **Melhor caso:**
  - O registro procurado é o **primeiro** consultado!!!
  - $f(n) = 1$.
- **Pior caso:**
  - O registro procurado é o **último** consultado!!!
  - $f(n) = n$.

Exemplo: Registros de um arquivo (Análise)

- **Melhor caso:**
  - O registro procurado é o **primeiro** consultado!!!
  - $f(n) = 1$.
- **Pior caso:**
  - O registro procurado é o **último** consultado!!!
  - $f(n) = n$.
- **Caso médio:**

Exemplo: Registros de um arquivo (Análise)

- **Melhor caso:**
  - O registro procurado é o **primeiro** consultado!!!
  - $f(n) = 1$.
- **Pior caso:**
  - O registro procurado é o **último** consultado!!!
  - $f(n) = n$.
- **Caso médio:**
  - $f(n) = (n+1)/2$.
  - *Considerando a mesma probabilidade de acesso para todos os registros.*

Exemplo: Maior e menor elementos de um vetor

- Considere o problema de encontrar o maior e o menor elementos de um vetor de inteiros $A[n]$, onde $n >= 1$.
- Um algoritmo simples pode ser derivado do algoritmo apresentado no programa para achar o **maior** elemento:

```
int Max(int* A, int n) {
  int i, Temp;
  Temp = A[0];
  for(i = 1; i < n; i++)
    if(Temp < A[i]) // Comparação envolvendo os elem.
      Temp = A[i];
  return Temp;
}
```

  - Recordando que, para `Max`: $f(n) = n - 1$.

Exemplo: Maior e menor elementos de um vetor

- Considere o problema de encontrar o maior e o menor elementos de um vetor de inteiros $A[n]$, onde $n >= 1$.
- Um algoritmo simples pode ser derivado do algoritmo apresentado no programa para achar o **maior** elemento:

```
int Max(int* A, int n) {
  int i, Temp;
  Temp = A[0];
  for(i = 1; i < n; i++)
    if(Temp < A[i]) // Comparação envolvendo os elem.
      Temp = A[i];
  return Temp;
}
```

  - Recordando que, para `Max`: $f(n) = n - 1$.

Exemplo: Maior e menor elementos de um vetor (MaxMin1)

```c
void MaxMin1(int* A, int n, int* pMax, int* pMin) {
  int i;
  *pMax = A[0];
  *pMin = A[0];
  for(i = 1; i < n; i++) {
    if(*pMax < A[i]) // Comparação envolvendo os elementos
      *pMax = A[i];
    if(*pMin > A[i]) // Comparação envolvendo os elementos
      *pMin = A[i];
  }
}
```

Exemplo: Maior e menor elementos de um vetor (MaxMin1)

```c
void MaxMin1(int* A, int n, int* pMax, int* pMin) {
  int i;
  *pMax = A[0];
  *pMin = A[0];
  for(i = 1; i < n; i++) {
    if(*pMax < A[i]) // Comparação envolvendo os elementos
      *pMax = A[i];
    if(*pMin > A[i]) // Comparação envolvendo os elementos
      *pMin = A[i];
  }
}
```

- Seja $f(n)$ o número de comparações entre os elementos de A, se A contiver n elementos.

Exemplo: Maior e menor elementos de um vetor (MaxMin1)

```
void MaxMin1(int* A, int n, int* pMax, int* pMin) {
  int i;
  *pMax = A[0];
  *pMin = A[0];
  for(i = 1; i < n; i++) {
    if(*pMax < A[i]) // Comparação envolvendo os elementos
      *pMax = A[i];
    if(*pMin > A[i]) // Comparação envolvendo os elementos
      *pMin = A[i];
  }
}
```

- Seja $f(n)$ o número de comparações entre os elementos de A, se A contiver n elementos.
- Então, $f(n) = 2(n-1)$ para $n > 0$, para o melhor caso, pior caso e caso médio.

Exemplo: Maior e menor elementos de um vetor (MaxMin2)

- `MaxMin1` pode ser facilmente melhorado:
  - A comparação $*pMin > A[i]$ só é necessária quando a comparação $*pMax < A[i]$ é falsa.

```c
void MaxMin2(int* A, int n, int* pMax, int* pMin) {
  int i;
  *pMax = A[0];
  *pMin = A[0];
  for(i = 1; i < n; i++)
    if(*pMax < A[i])
      *pMax = A[i];
    else if(*pMin > A[i]) // A diferença está aqui!
      *pMin = A[i];
}
```

- E agora, qual é a função de complexidade?

Exemplo: Maior e menor elementos de um vetor (MaxMin2)

- MaxMin1 pode ser facilmente melhorado:
  - A comparação $*pMin > A[i]$ só é necessária quando a comparação $*pMax < A[i]$ é falsa.

```c
void MaxMin2(int* A, int n, int* pMax, int* pMin) {
  int i;
  *pMax = A[0];
  *pMin = A[0];
  for(i = 1; i < n; i++)
    if(*pMax < A[i])
      *pMax = A[i];
    else if(*pMin > A[i]) // A diferença está aqui!
      *pMin = A[i];
}
```

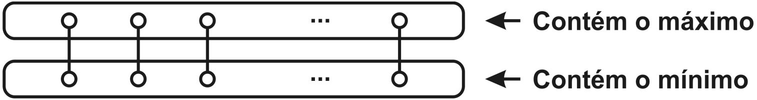
- **E agora, qual é a função de complexidade?**

Exemplo: Maior e menor elementos de um vetor (MaxMin2)

- **Melhor caso:**
  - Quando os elementos estão em ordem crescente.
  - $f(n) = n - 1$.
- **Pior caso:**
  - Quando o maior elemento é o primeiro no vetor.
  - $f(n) = 2(n - 1)$.
- **Caso médio:**
  - Quando $*pMax < A[i]$ em metade das vezes.
  - $f(n) = 3n/2 - 3/2$.

Exemplo: Maior e menor elementos de um vetor (MaxMin3)

- Considerando o número de comparações realizadas, existe a possibilidade de obter um algoritmo mais eficiente:
  1. Compare os elementos de `A` aos pares, separando-os em **dois subconjuntos** (**maiores** em um e **menores** em outro), a um custo de $\lceil n/2 \rceil$ comparações.
  2. O **máximo** é obtido do subconjunto que contém os maiores elementos, a um custo de $\lceil n/2 \rceil - 1$ comparações.
  3. O **mínimo** é obtido do subconjunto que contém os menores elementos, a um custo de $\lceil n/2 \rceil - 1$ comparações.



Exemplo: Maior e menor elementos de um vetor (MaxMin3)

- Os elementos de `A` são comparados dois a dois.
  - Os elementos maiores são comparados com $*pMax$.
  - Os elementos menores são comparados com $*pMin$.
- Quando n é ímpar, o elemento que está na posição $A[n-1]$ é duplicado na posição $A[n]$ para evitar um tratamento de exceção.
- Para esta implementação:
  - **Melhor caso = Pior caso = Caso médio:**

$$f(n) = \frac{n}{2} + \frac{n-2}{2} + \frac{n-2}{2} = \frac{3n}{2} - 2$$

Exemplo: Maior e menor elementos de um vetor (MaxMin3)

```c
void MaxMin3(int* A, int n, int* pMax, int* pMin) {
  int i, FimDoAnel;
  if((n % 2) > 0) { A[n] = A[n-1]; FimDoAnel = n; }
  else            { FimDoAnel = n-1;              }
  if(A[0] > A[1]) { *pMax = A[0]; *pMin = A[1]; } // Comp. 1
  else            { *pMax = A[1]; *pMin = A[0]; }

  for(i=2; i<FimDoAnel; i+=2) {
    if(A[i] > A[i+1]) {                           // Comp. 2
      if(A[i] > *pMax) *pMax = A[i];              // Comp. 3
      if(A[i+1] < *pMin) *pMin = A[i+1];          // Comp. 4
    } else {
      if(A[i] < *pMin) *pMin = A[i];              // Comp. 3
      if(A[i+1] > *pMax) *pMax = A[i+1];          // Comp. 4
    }
  }
}
```

Exemplo: Maior e menor elementos de um vetor (MaxMin3)

- **Qual a função de complexidade?**
- Número de comparações:
  - `Comp.1`: $\mathbf{1}$ vez.
  - `Comp.2`: $\boldsymbol{n/2-1}$ vezes.
  - `Comp.3`: $\boldsymbol{n/2-1}$ vezes.
  - `Comp.4`: $\boldsymbol{n/2-1}$ vezes.
- Então:
  $f(n) = 1 + (n/2-1) + (n/2-1) + (n/2-1)$
  $f(n) = 3(n/2-1) + 1$
  $f(n) = 3n/2 - 2$

Exemplo: Maior e menor elementos de um vetor (comparação)

Algoritmo	Melhor caso	Pior caso	Caso médio
MaxMin1	2 (n - 1)	2 (n - 1)	2 (n - 1)
MaxMin2	n - 1	2 (n - 1)	3 n/2 - 3/2
MaxMin3	3 n/2 - 2	3 n/2 - 2	3 n/2 - 2

- Os algoritmos `MaxMin2` e `MaxMin3` são superiores ao algoritmo `MaxMin1`.
- O algoritmo `MaxMin3` é superior ao algoritmo `MaxMin2` com relação ao pior caso e muito próximo quanto ao caso médio.

Exemplo: Maior e menor elementos de um vetor (comparação)

Algoritmo	Melhor caso	Pior caso	Caso médio
MaxMin1	2 (n - 1)	2 (n - 1)	2 (n - 1)
MaxMin2	n - 1	2 (n - 1)	3 n/2 - 3/2
MaxMin3	3 n/2 - 2	3 n/2 - 2	3 n/2 - 2

- Os algoritmos `MaxMin2` e `MaxMin3` são superiores ao algoritmo `MaxMin1`.
- O algoritmo `MaxMin3` é superior ao algoritmo `MaxMin2` com relação ao pior caso e muito próximo quanto ao caso médio.

Conteúdo

Conclusão

- Nesta aula tivemos o primeiro contato com a **Análise de Complexidade de Algoritmos**.
- Este tópico é muito importante no estudo de algoritmos, pois possibilita avaliar o **custo** dos mesmos, servindo de referência para a escolha daquele que será mais adequado para uso em determinadas situações.
- Foco principal para a definição da **função de complexidade** e a análise de **melhor caso**, **pior caso** e **caso médio**.
- *Próxima aula*: Análise de Algoritmos (Parte 2) – Comportamento Assintótico e Dominação Assintótica.
- Dúvidas?

Conclusão

- Nesta aula tivemos o primeiro contato com a **Análise de Complexidade de Algoritmos**.
- Este tópico é muito importante no estudo de algoritmos, pois possibilita avaliar o **custo** dos mesmos, servindo de referência para a escolha daquele que será mais adequado para uso em determinadas situações.
- Foco principal para a definição da **função de complexidade** e a análise de **melhor caso**, **pior caso** e **caso médio**.
- *Próxima aula*: Análise de Algoritmos (Parte 2) – Comportamento Assintótico e Dominação Assintótica.
- Dúvidas?

Conclusão

- Nesta aula tivemos o primeiro contato com a **Análise de Complexidade de Algoritmos**.
- Este tópico é muito importante no estudo de algoritmos, pois possibilita avaliar o **custo** dos mesmos, servindo de referência para a escolha daquele que será mais adequado para uso em determinadas situações.
- Foco principal para a definição da **função de complexidade** e a análise de **melhor caso**, **pior caso** e **caso médio**.
- *Próxima aula*: Análise de Algoritmos (Parte 2) – Comportamento Assintótico e Dominação Assintótica.
- Dúvidas?

Conclusão

- Nesta aula tivemos o primeiro contato com a **Análise de Complexidade de Algoritmos**.
- Este tópico é muito importante no estudo de algoritmos, pois possibilita avaliar o **custo** dos mesmos, servindo de referência para a escolha daquele que será mais adequado para uso em determinadas situações.
- Foco principal para a definição da **função de complexidade** e a análise de **melhor caso**, **pior caso** e **caso médio**.
- *Próxima aula*: Análise de Algoritmos (Parte 2) – Comportamento Assintótico e Dominação Assintótica.
- Dúvidas?

Conclusão

- Nesta aula tivemos o primeiro contato com a **Análise de Complexidade de Algoritmos**.
- Este tópico é muito importante no estudo de algoritmos, pois possibilita avaliar o **custo** dos mesmos, servindo de referência para a escolha daquele que será mais adequado para uso em determinadas situações.
- Foco principal para a definição da **função de complexidade** e a análise de **melhor caso**, **pior caso** e **caso médio**.
- *Próxima aula*: Análise de Algoritmos (Parte 2) – Comportamento Assintótico e Dominação Assintótica.
- Dúvidas?

Conteúdo

Exercício 01

- Avalie os dois códigos fonte abaixo e responda:
  - O resultado será o mesmo? Justifique sua resposta.
  - Qual a função de complexidade de cada um dos procedimentos? Defina as operações relevantes.
  - Caso o resultado seja o mesmo, qual dos dois você escolheria?

```c
void Procedimento1() {
  int i = 0, a = 0;
  while(i < n) {
    a += i;
    i += 2;
  }
}
```

```c
void Procedimento2() {
  int i, j, a = 0;
  for(i = 0; i < n; i++)
    for(j = 0; j < i; j++)
      a += i + j;
}
```

Dica: Avalie o código e faça testes de mesa, só depois de responder às perguntas, implemente o código e execute os procedimentos para conferir os resultados.