Estrutura de Dados e Algoritmos

Análise de Complexidade de Códigos

- Definir uma fórmula que expresse o tempo de execução de um algoritmo em função do volume de dados de entrada;
- Estimativa de tempo de execução;

Regras para Análise de Algoritmos

- Instruções Simples:

```
int x;
float y;
x = 1;
y = 1.0 + x;
```

Podemos facilmente notar que qualquer desses comandos, a não ser que esteja dentro de um laço, será executado uma única vez. Ou seja, **são comandos que não dependem do volume de dados** de entrada. **Portanto, dizemos que esses comandos têm ordem de complexidade constante, ou O(1).**

Regras para Análise de Algoritmos

- Laços de Repetição (1):

a

b

c: i=i+1 → =, +

```
for (i=0;i<10;i++)
{
  vetor[i] = i;
  vetor[i+1] = x;
}
```

d

e

f

Como o tempo é constante, isto é, independente do volume de dados de entrada, podemos simplificar e dizer que a complexidade é simplesmente **O(1)**.

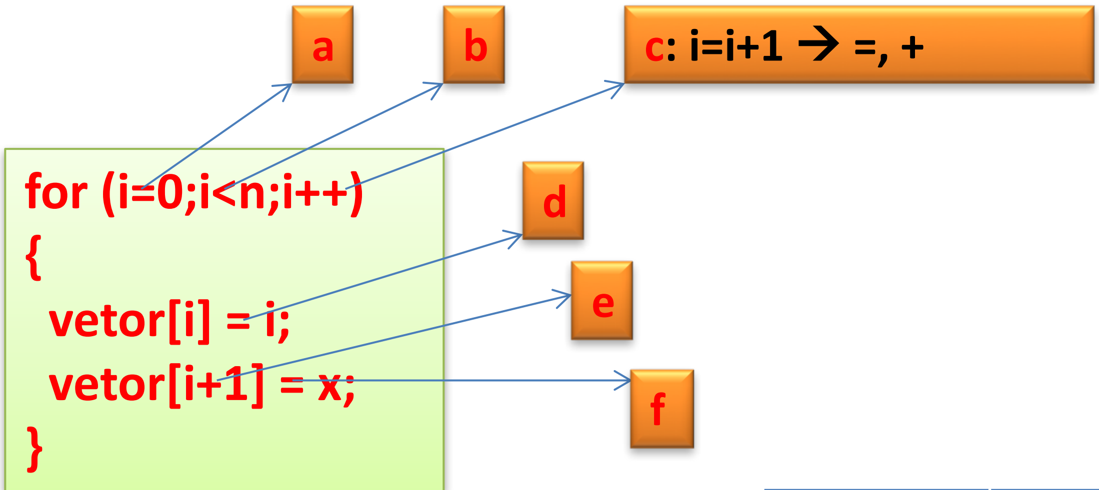
Instrução	Qtde de Execuções	O()
a	1	O(1)
b	11	O(11)
c	10	O(20)
d	10	O(10)

Instrução	Qtde de Execuções	O()
e	10	O(10)
f	10	O(10)

O(1)+O(11) +O(20) +O(10) +O(10) +O(10) =
O(1+11+20+10+10+10) = **O(62)**

Regras para Análise de Algoritmos

- Laços de Repetição (2):



Instrução	Qtde de Execuções	O()
a	1	O(1)
b	n+1	O(n+1)
c	n	O(2n)
d	n	O(n)

Instrução	Qtde de Execuções	O()
e	n	O(n)
f	n	O(n)

$O(1) + O(n+1) + O(2n) + O(n) + O(n) + O(n) =$
$O(1 + n+1 + 2n + n + n + n) = O(6n+2) \rightarrow$ **O(n)**

Regras para Análise de Algoritmos

- Laços Aninhados:

```
for (i=0;i<n;i++)
{
  for (j=0;j<n;j++)
  {
    k = k + i*j;
  }
}
```

a

b

c: $i=i+1 \rightarrow =, +$

d

e

f: $i=i+1 \rightarrow =, +$

g

h

i

$O(1) + O(n+1) + O(2n+2) + O(n) + O(n^2) + O(2n^2+2n) + O(n^2) + O(n^2) + O(n^2) =$
$O(1 + n+1 + 2n+2 + n + n^2 + 2n^2+2n + n^2 + n^2 + n^2) =$
$O(6n^2+6n+4) \rightarrow O(n^2)$

Instrução	Qtde de Execuções	O()
a	1	$O(1)$
b	$n+1$	$O(n+1)$
c	$n+1$	$O(2n+2)$
d	n	$O(n)$

Instrução	Qtde de Execuções	O()
e	$n.(n+1) \rightarrow n^2 + n$	$O(n^2)$
f	$n.(n+1) \rightarrow n^2 + n$	$O(2n^2 +2n)$
g	n^2	$O(n^2)$
h	n^2	$O(n^2)$
i	n^2	$O(n^2)$

UnB GAMA

Regras para Análise de Algoritmos

Universidade de Brasília - UnB

- Estruturas Consecutivas:

```
k = n*4 – pow(7,2);
for (i=0;i<10;i++)
{
  vetor[i] = i;
  vetor[i+1] = x;
}
for (i=0;i<n;i++)
{
  vetor[i] = i;
  vetor[i+1] = x;
}
for (i=0;i<n;i++)
{
  for (j=0;j<n;j++)
  {
    k = k + i*j;
  }
}
```

O(1)

O(1)

O(n)

$O(n^2)$

$O(1) + O(n) + O(n^2) = O(1 + n + n^2) \rightarrow \mathbf{O(n^2)}$

Regras para Análise de Algoritmos

- Estrutura Condicional:

```
If condição            → O(1)
{
  Instrução 1;
  ...                  → O(f(n))
  Instruçao c;
}
else
{
  Instrução c+1;
  ...                  → O(g(n))
  Instruçao k;
}
```

= O(1) + O(f(n)) = O(1 + f(n)), se condiçao é verdadeira
= O(1) + O(g(n)) = O(1 + g(n)), se condiçao é falsa

Regras para Análise de Algoritmos

- Funções:

Para calcular a complexidade com chamadas a funções, calcule primeiramente a complexidade de cada uma das funções e depois considere cada uma das funções como uma instrução, com a complexidade da função.

```
pvf1 = f1(n);      → O(f1)
.
.
pvf2 = f2(3);      → O(f2)
.
.
pvf3 = f3(n,n);    → O(f3)
```

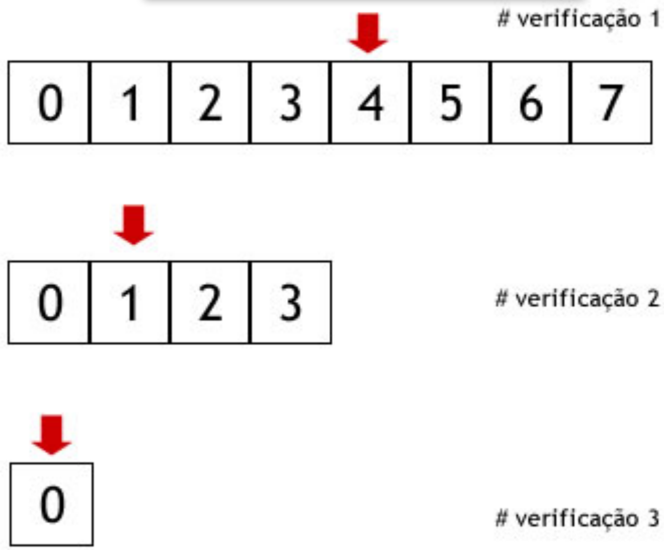
O(f1) + O(f2) + O(f3) → ***max(O(f1), O(f2), O(f3))***

Regras para Análise de Algoritmos

- Algoritmos $O(log_2 n)$:

```
int lo = 0, mid, hi = n - 1;
while (lo <= hi)
{
  mid = (lo + hi)/2;
  if (key < a[mid])
    hi = mid - 1;
  else if (key > a[mid])
    lo = mid + 1;
  else
    return mid;
}
return -1;
```

Exemplo para n = 8



Supondo que a chave não esteja no vetor, o laço é executado m vezes, onde $2^m = n$. Ou: $m = log_2 n \rightarrow$ **$O(log_2 n)$**.

Regras para Análise de Algoritmos

- **Quando o tempo de resposta de um algoritmo depende também da configuração dos dados de entrada, podemos ter as seguintes análises:**
  - **Melhor caso:** menor tempo de execução;
  - **Pior caso:** maior tempo de execução. Geralmente, priorizamos determinar o pior caso;
  - **Caso médio:** média dos tempos de execução. Mais difícil de obter.

- Exemplo: busca sequencial.
- Operação: comparação de x com elementos de V;

```
buscaSequencial(x,V)
i = 0;
enquanto (i < n) e (V[i] != x) // executa n vezes no máximo
{
  i  = i + 1;
}
se i >= n então “Busca sem sucesso”
senão “Busca com sucesso”
```

- Melhor caso da busca sequencial: x está em V[0].
- Complexidade de tempo: O(1).

```
buscaSequencial(x,V)
i = 0;
enquanto (i < n) e (V[i] != x) // executa n vezes no máximo
{
  i  = i + 1;
}
se i >= n então "Busca sem sucesso"
senão "Busca com sucesso"
```

- Pior caso da busca sequencial: x está em V[n-1] ou não está em V.
- Complexidade de tempo: O(n).

```
buscaSequencial(x,V)
i = 0;
enquanto (i < n) e (V[i] != x) // executa n vezes no máximo
{
  i  = i + 1;
}
se i >= n então “Busca sem sucesso”
senão “Busca com sucesso”
```

- Pior caso da busca sequencial: x está em V[n-1] ou não está em V.
- Complexidade de tempo: O(n).

```
buscaSequencial(x,V)
i = 0;
enquanto (i < n) e (V[i] != x) // executa n vezes no máximo
{
  i  = i + 1;
}
se i >= n então “Busca sem sucesso”
senão “Busca com sucesso”
```

Regras para Análise de Algoritmos

- Caso médio da busca sequencial: assumindo que x está em V, $f(n) = 1 \times p1 + 2 \times p2 + \ldots + n \times pn$ onde pi é probabilidade de x estar na posição i;
- Considerando que as probabilidades são iguais: $pi = 1/n$.
- $f(n) = 1/n(1 + 2 + 3 + \ldots + n) = 1/n(n(n+1)/2) = (n+1)/2$
- Complexidade de tempo: $(n+1)/2$, ou seja, uma pesquisa bem-sucedida examina aproximadamente metade dos registros.
- A complexidade do Caso médido é $O(n)$.